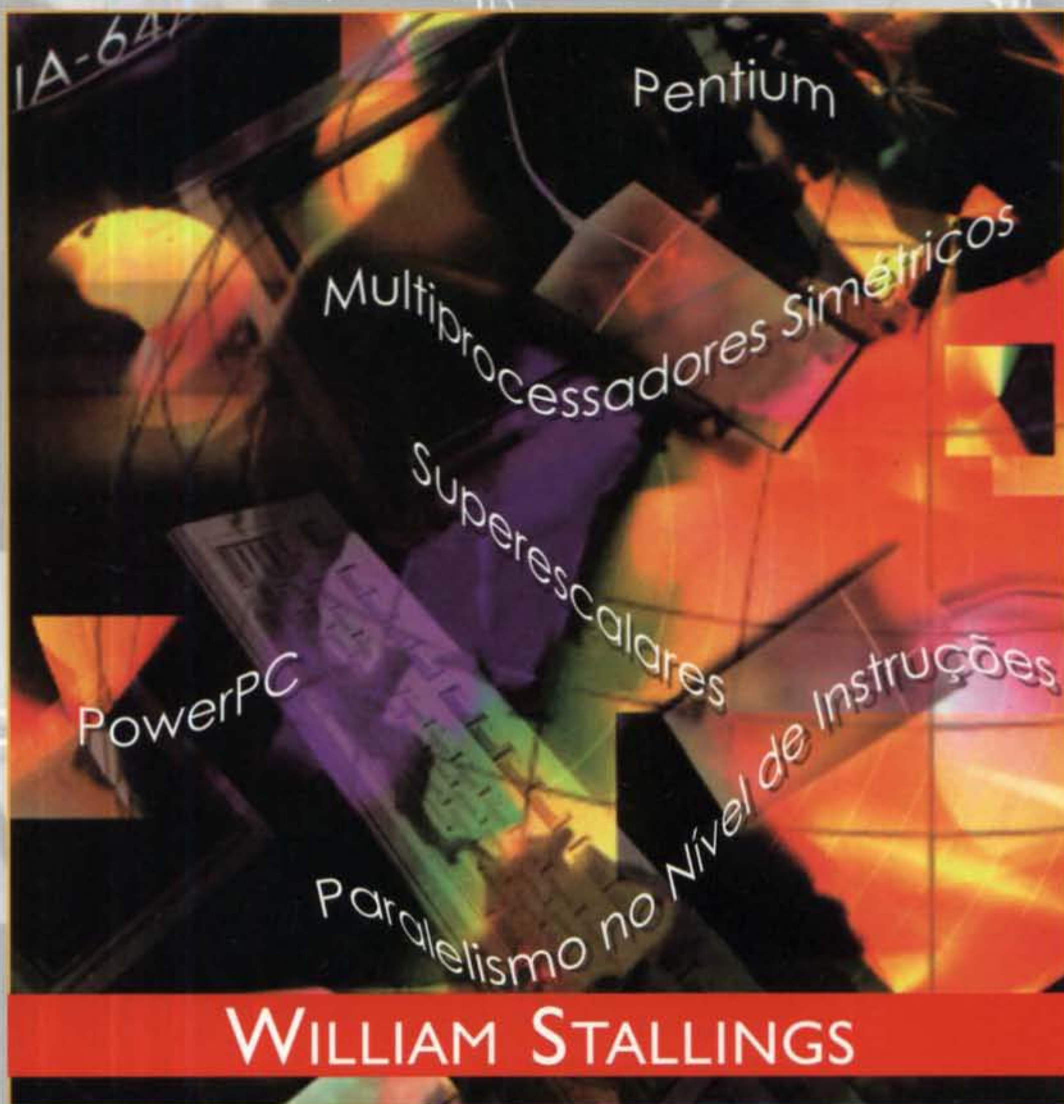
Arquitetura e Organização de Computadores

QUINTA EDIÇÃO



WILLIAM STALLINGS

Prentice Hall

Arquitetura e Organização de Computadores

Projeto para o Desempenho

Quinta Edição

WILLIAM STALLINGS

Tradução
Carlos Camarão de Figueiredo
Doutor em Ciência da Computação pela Universidade de Manchester – Inglaterra
Lucília Camarão de Figueiredo
Doutora em Ciência da Computação pela Pontifícia Universidade Católica - RJ
Professora do Departamento de Computação da Universidade Federal de Ouro Preto - MG

Revisão Técnica
Edson Toshimi Midorikawa
Professor Doutor do Departamento de Engenharia de Computação e
Sistemas Digitais da Escola Politécnica da Universidade de São Paulo





São Paulo – 2003

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Porto Rico Venezuela

Editor: Roger Trimer
Produtora Editorial: Renatha Prado
Capa: Marcelo da Silva Françozo, sobre o projeto original de Heather Scott
Editoração Eletrônica: ERJ Composição Editorial e Artes Gráficas Ltda.
Impressão: São Paulo – SP

Dados de Catalogação na Publicação

Stallings, William
Arquitetura e Organização de Computadores:
Projeto para o Desempenho.

William Stallings;
tradução: Carlos Camarão de Figueiredo e Lucília Camarão de Figueiredo;
revisão técnica: Edson Toshimi Midorikawa.
—São Paulo: Prentice Hall, 2002

ISBN: 85.87918-53-2

Índices para catalogação:

1. Computadores: arquitetura e organização
2. Arquitetura de computadores

2003
Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil,
uma empresa do grupo Pearson Education
Rua Emílio Goeldi, 747
CEP 05065-110, São Paulo – SP, Brasil
Fone (11) 3613-1222 Fax (11) 3611-0444
e-mail: vendas@pearsoned.com.br

Para minha generosa mulher
ATS
e para as suas constantes companhias,
Geoffroi e Princesa Kate Lan Kinetic,
Les Enfants du Paradis

PREFÁCIO

Objetivos

Este livro trata da estrutura e do funcionamento de computadores. Seu objetivo é apresentar, da forma mais clara e abrangente possível, a natureza e as características dos sistemas de computação modernos.

Isso constitui tarefa desafiadora, por várias razões. Em primeiro lugar, existe uma enorme variedade de produtos que podem ser denominados 'computadores', desde processadores de um único *chip* ou pastilha, que custam poucos dólares, até supercomputadores, que custam dezenas de milhões de dólares. Essa variedade apresenta-se não apenas em relação ao custo, mas também em relação ao tamanho, ao desempenho e à aplicação. Em segundo lugar, a rápida evolução que sempre caracterizou a tecnologia de computadores continua sem limites. Essa evolução engloba todos os aspectos da tecnologia de computadores, desde a tecnologia de circuitos integrados usados na construção dos seus componentes até a crescente utilização de conceitos de organização paralela na combinação desses componentes.

Apesar da grande variedade e da rapidez de evolução da área de computação, certos conceitos fundamentais aplicam-se a qualquer projeto de computadores. A aplicação desses conceitos depende do estado atual da tecnologia e dos objetivos de custo e de desempenho do projetista. A intenção deste livro é oferecer uma discussão minuciosa sobre os conceitos fundamentais de arquitetura e organização de computadores, relacionando-os com as questões de projeto modernas.

O subtítulo sugere o tema e a abordagem adotados neste livro. Sempre foi importante projetar computadores com grande desempenho, mas essa exigência nunca foi tão forte e tão difícil de ser atendida como nos dias de hoje. Todas as características básicas de desempenho de sistemas de computação, como velocidade do processador, velocidade da memória, capacidade de armazenamento da memória e taxas de transmissão de dados, têm crescido rapidamente. Além disso, esse crescimento ocorre a taxas diferentes. Isso dificulta o projeto de um sistema balanceado, que maximize tanto o desempenho quanto a utilização de todos os elementos do sistema. O projeto de computadores vem se tornando, portanto, cada vez mais, um jogo de alterar a estrutura ou a função em uma determinada área para compensar um mau desempenho em outra área. Esse jogo poderá ser observado em inúmeras decisões de projeto apresentadas ao longo deste livro.

Um sistema de computação, como qualquer outro sistema, consiste de um conjunto de componentes inter-relacionados. Um sistema é mais bem caracterizado em termos da sua estrutura, o modo como os componentes estão interconectados, e do seu funcionamento — a operação de seus componentes individuais. Além disso, a organização de um computador é

hierárquica: cada componente principal pode ser descrito pela sua decomposição em subcomponentes, juntamente com a descrição da estrutura e do funcionamento desses componentes. Para maior clareza e facilidade de entendimento, essa organização hierárquica é apresentada, neste livro, do nível mais alto para o mais baixo:

- **Sistema de computação:** Seus principais componentes são o processador, a memória e os dispositivos de E/S.
- **Processador**: Seus principais componentes são a unidade de controle, os registradores, a unidade lógica e aritmética (ULA) e a unidade de execução de instruções.
- **Unidade de controle**: Seus principais componentes são a memória de controle, a lógica de seqüenciamento de microinstruções e os registradores.

O objetivo é apresentar o material do livro de maneira que introduza cada novo conceito no contexto em que ele se aplica. Isso deverá minimizar o risco de o leitor se perder ao longo do texto, além de motivá-lo de forma mais adequada do que em uma abordagem na qual a organização hierárquica é apresentada do nível mais baixo para o mais alto.

Ao longo da discussão, vários aspectos do sistema serão abordados, tanto do ponto de vista da sua arquitetura (os atributos do sistema que são visíveis para um programador de programas em linguagem de máquina) como do ponto de vista da sua organização (as unidades operacionais e suas interconexões que realizam a arquitetura).

Exemplos de Sistemas

Ao longo deste livro, serão usados exemplos de diferentes máquinas, para esclarecer e reforçar os conceitos apresentados. A maioria dos exemplos é extraída de duas famílias de computadores: o Pentium II da Intel e o PowerPC. (O Pentium III, recentemente introduzido no mercado, é, essencialmente, o Pentium II com um conjunto expandido de instruções multimídia.) Juntos, os projetos desses dois sistemas seguem as tendências da maioria dos projetos atuais de computadores. O Pentium II é, fundamentalmente, um computador que possui um conjunto complexo de instruções (CISC), embora com um núcleo RISC; o PowerPC é essencialmente um computador com um conjunto reduzido de instruções (RISC). Ambos fazem uso de princípios de projeto de arquitetura superescalar e provêem suporte a configurações com múltiplos processadores.*

Organização do Texto

O livro é organizado em cinco partes:

Parte 1 Visão geral: Essa parte oferece uma visão geral do restante do livro.

Parte 2 O sistema de computação: Um sistema de computação é constituído de processador, memória e módulos de E/S, além das interconexões entre esses componentes principais.

* N.R.T.: O novo processador da Intel, o Pentium 4, apresenta as mesmas características dos seus antecessores, como, por exemplo, a implementação da arquitetura IA-32, acrescentando a extensão SSE2 (Streaming SIMD Extensions 2) no conjunto de instruções e uma nova microarquitetura, chamada NetBurst. Para maiores informações consulte a referência "IA-32 Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture" disponível na página Web da Intel (http://www.intel.com).

Com exceção do processador, que é suficientemente complexo para ser explorado na Parte 3, essa parte aborda cada um dos demais componentes.

Parte 3 A unidade de processamento central: A CPU consiste de uma unidade de controle, registradores, unidade lógica e aritmética, unidade de execução de instruções e interconexões entre esses componentes. Nessa parte, são abordados aspectos da arquitetura da CPU, tais como o projeto do conjunto de instruções e dos tipos de dados. Também são tratadas questões relativas à sua organização, como, por exemplo, o uso de *pipeline*.

Parte 4 A unidade de controle: A unidade de controle é o componente do processador que ativa os demais componentes. Essa parte trata do funcionamento da unidade de controle e da sua implementação, utilizando microprogramação.

Parte 5 Organização paralela: Essa última parte aborda algumas das questões envolvidas em organizações com múltiplos processadores e com processamento vetorial.

Um resumo mais detalhado do conteúdo do livro, com uma descrição do conteúdo de cada capítulo, é apresentado no final do Capítulo 1.

Projetos para Ensino de Arquitetura e Organização de Computadores

Para muitos professores, um componente importante de um curso de arquitetura e organização de computadores é o desenvolvimento de um projeto, ou de um conjunto de projetos, por meio dos quais os estudantes possam pôr em prática e reforçar o aprendizado dos conceitos introduzidos no texto. Este livro inclui material adicional para prover suporte adequado ao desenvolvimento de projetos ao longo do curso. O manual do professor não apenas contém orientação sobre como definir e estruturar os projetos, mas também sugere um conjunto de projetos, que cobrem diversos tópicos do texto:

- **Projetos de pesquisa**: O manual inclui uma série de atividades de pesquisa, que instruem o estudante a realizar uma pesquisa sobre um determinado tópico na Web ou na literatura técnica e a escrever um relatório.
- **Projetos de simulação**: O manual fornece informações necessárias para a utilização do pacote de simulação SimpleScalar, que pode ser utilizado para explorar aspectos relativos ao projeto da arquitetura e da organização de um computador.
- **Atividades de leitura e relatório**: O manual inclui uma relação de artigos sugeridos como leitura complementar, um ou mais para cada capítulo, cuja leitura pode ser atribuída como atividade para o estudante, propondo-se que ele escreva um breve resumo sobre cada artigo lido.

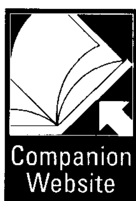
Veja o Apêndice B para maiores detalhes.

O Que Há de Novo na Quinta Edição

Nos quatro anos decorridos desde o lançamento da quarta edição deste livro, contínuas inovações e melhorias ocorreram na área. Nesta nova edição, procuramos cobrir essa evolução, mantendo, ao mesmo tempo, uma abordagem clara e abrangente da área. Para iniciar esse processo de revisão, a quarta edição deste livro foi exaustivamente revisada por inúmeros professores que lecionam o assunto. Como resultado, a apresentação tornou-se mais clara e concisa, e várias ilustrações foram aprimoradas. Além disso, foram adicionados diversos exercícios novos, previamente testados em sala de aula.

Além dessas melhorias introduzidas com objetivos pedagógicos e para tornar o livro mais agradável ao estudante, outras modificações substanciais foram feitas. Embora a estrutura de capítulos tenha se mantido praticamente inalterada, a maior parte do conteúdo de cada capítulo foi revisada e novos materiais foram incluídos. Algumas das principais mudanças são:

- **Memória óptica**: O material referente à memória óptica foi ampliado, passando a incluir dispositivos de memória magneto-óptica.
- **Projeto de processador superescalar**: O capítulo relativo a projeto de processadores superescalares foi ampliado, incluindo uma discussão mais detalhada e dois novos exemplos: o UltraSparc II e o MIPS R10000.
- **Conjunto de instruções multimídia**: O conjunto de instruções MMX, usado no Pentium II e no Pentium III, é abordado.
- **Execução predicativa e carga especulativa**: Esta edição realça a discussão desses dois conceitos recentes, que são fundamentais no projeto da nova arquitetura IA-64 da Intel e da Hewlett-Packard.
- **SMPs, *clusters* e sistemas NUMA**: O capítulo relativo à organização paralela de computadores foi totalmente reescrito. O novo capítulo inclui a descrição detalhada e a comparação entre multiprocessadores simétricos (SMPs), *clusters* e sistemas com acesso não-uniforme à memória (NUMA).
- **Material expandido para o professor**: Como foi dito anteriormente, o livro agora oferece extenso material para prover suporte a projetos. Foi também ampliado o material de apoio contido na página Web do livro.



Recursos On-Line para Arquitetura e Organização de Computadores

O Companion Website do livro (www.prenhall.com/stallings_br) oferece recursos adicionais para alunos e professores.

Manual de Soluções em Português

Os professores que adotam o livro têm acesso ao manual de soluções completo, disponível no site para download.
Elaborado pelo próprio autor, o manual foi traduzido pelo professor Edson Toshimi Midorikawa, da Escola Politécnica da Universidade de São Paulo.

Site do Autor

O Companion Website traz um link para o site do autor, onde estão disponíveis, em inglês, os seguintes recursos:

Materiais de Apoio ao Curso

- Cópias das figuras do livro em formato PDF.
- Notas de aula em formato PDF, adequadas como resumo para o estudante ou como um esquema global do curso.
- Slides para apoio ao ensino, elaborados no PowerPoint.

Páginas Web Úteis

A página do autor inclui endereços on-line relevantes, organizados por capítulos. Essas páginas cobrem um vasto espectro de tópicos, possibilitando aos estudantes explorar cada tópico com maior profundidade, no momento adequado.

Lista de Discussão na Internet

Uma lista de discussão, mantida na Internet, possibilita aos professores trocar informações, sugestões e perguntas relativas a este livro, entre si e com o autor.

Ferramentas de Simulação

Link para a página referente à ferramenta de simulação SimpleScalar, usada para a análise de decisões de projeto de processadores. Essa página contém tanto o software como as informações de suporte necessárias. O manual do professor inclui informações adicionais sobre instalação e utilização do software, assim como sugestões de projetos a serem executados pelos estudantes. O Apêndice B traz mais informações.

Agradecimentos

Esta nova edição beneficiou-se da revisão de inúmeras pessoas, que generosamente cederam seu tempo e conhecimento. As seguintes pessoas revisaram a quarta edição e fizeram diversas sugestões úteis: Kitty Niles e Yew Pen-Chung, da Universidade de Minnesota; Yuval Tamir, da UCLA; Arthur Werbner; Bina Ramamurthy, da SUNY Buffalo; e Marcus Gonçalves, da Automation Research Corp. David Lambert, da Intel, revisou o material referente ao Pentium. As seguintes pessoas revisaram partes do manuscrito da quinta edição: Jay Kubicky; Mike Albaugh, da Atari Games; Tom Callaway, da Silicon Graphics; James Stine, da Lehigh University; Gabriel dos Reis, da Ecole Normale Superieure de Cachan; e Rick Thomas, da Rutgers. Bernard Leppla, da IBM da Alemanha, auxiliou-me na compreensão da estratégia SMP do computador de grande porte da IBM. Cindy Norris, da Appalachian State University, contribuiu com a proposição de alguns exercícios. Todd Bezenek, da University of Wisconsin, e James Stine, da LeHigh University, prepararam os exercícios relativos ao SimpleScalar, contidos no manual do professor, e Todd preparou, também, o Guia do Usuário do SimpleScalar.

Os editores da edição em português agradecem aos professores Geraldo Lino de Campos e Wagner Zucchi, da Universidade de São Paulo, que avaliaram o livro e colaboraram em sua publicação.

Sumário

PARTE

1

VISÃO GERAL

OBJETIVOS

O propósito da Parte 1 é fornecer embasamento e contexto para o restante deste livro, apresentando os conceitos fundamentais de arquitetura e organização de computadores.

ROTEIRO

Capítulo 1: Introdução

O Capítulo 1 introduz o conceito de computador como um sistema hierárquico. Um computador pode ser visto como um sistema formado por um conjunto estruturado de componentes, e sua função pode ser compreendida em termos das funções desses componentes. Cada componente, por sua vez, pode ser descrito em termos de sua estrutura e função internas. Os níveis mais altos dessa visão hierárquica são abordados neste capítulo. O restante do livro é organizado segundo essa estrutura hierárquica, em ordem decrescente.

Capítulo 2: Evolução e desempenho de computadores

O Capítulo 2 apresenta um breve histórico da evolução dos computadores, desde os seus ancestrais mecânicos até os sistemas atuais. Esse histórico contribui para destacar algumas características importantes do projeto de computadores e para dar uma visão de alto nível da estrutura de um computador. Em seguida, o capítulo introduz um tema fundamental deste livro: o projeto de computadores que visa a um melhor desempenho. Destaca-se, ainda, a importância de obter um balanceamento adequado da utilização dos diversos componentes de um computador, cujas características de desempenho são bastante distintas.

capítulo

1

INTRODUÇÃO

1.1 Arquitetura e organização

1.2 Estrutura e função

Função

Estrutura

1.3 Estrutura do livro

Evolução e desempenho de computadores

Barramentos do sistema

Memória interna

Memória externa

Entrada e saída

Suporte ao sistema operacional

Aritmética computacional

Conjuntos de instruções

Estrutura e funcionamento da CPU

Computadores RISC

Paralelismo em nível de instrução e processadores superescalares

Operação da unidade de controle

Controle microprogramado

Processamento paralelo

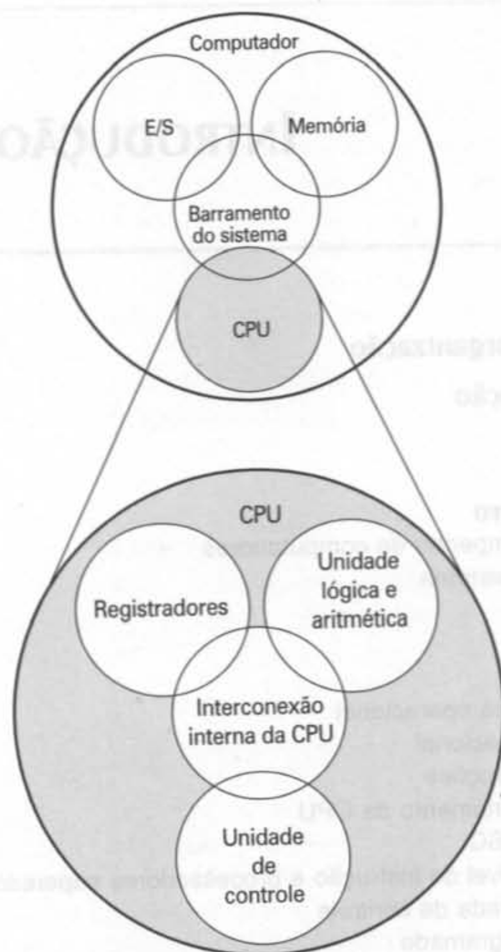
Lógica digital

1.4 Internet e recursos na WEB

Sites Web deste livro

Outros sites Web

Grupos de notícias USENET



- Os principais elementos de um sistema de computação são a unidade central de processamento (*central processing unit* — CPU), a memória principal, o subsistema de E/S (entrada e saída) e os mecanismos de interconexão entre esses componentes. A CPU, por sua vez, consiste em uma unidade de controle, uma unidade lógica e aritmética (*arithmetic and logic unit* — ULA), registradores internos e mecanismos de interconexão.
- Informações adicionais sobre este livro podem ser encontradas no seu site Web, que fornece endereços de outras páginas relevantes e outras informações úteis. Para maiores detalhes, veja a seção relativa a esse site Web no início deste livro.

Este livro trata da estrutura e da função de computadores. Seu objetivo é apresentar, da maneira mais clara e abrangente possível, a natureza e as características dos sistemas de computação modernos.

Isso constitui tarefa desafiadora, por várias razões. Em primeiro lugar, existe uma enorme variedade de produtos que podem ser denominados *computadores*, desde microcomputadores baseados em uma única pastilha (*chip*), que custam poucos dólares, até supercomputadores, no valor de dezenas de milhões de dólares. Essa variedade apresenta-se não apenas em relação ao custo, mas também em relação ao tamanho, ao desempenho e à aplicação. Em segundo lugar, a rápida evolução que sempre caracterizou a tecnologia de computadores continua sem limites. Essa evolução engloba todos os aspectos da tecnologia de computadores, desde a tecnologia de circuitos integrados usados na construção dos seus componentes até a crescente utilização de conceitos de organização paralela na combinação desses componentes.

Apesar da grande variedade e da rapidez da evolução da área, certos conceitos fundamentais aplicam-se a qualquer projeto de computadores. A aplicação desses conceitos depende do estado atual da tecnologia e dos requisitos de desempenho e de custo do projeto. O objetivo deste livro é fornecer uma discussão minuciosa sobre os conceitos fundamentais de arquitetura e organização de computadores, relacionando-os com as questões de projeto de computadores modernos. Este capítulo introduz a abordagem adotada para a estrutura deste livro e apresenta uma visão geral do restante do seu conteúdo.

1.1 ARQUITETURA E ORGANIZAÇÃO

Ao se descrever um sistema de computação, é feita uma distinção entre a *arquitetura* e a *organização do computador*. Embora seja difícil definir precisamente esses termos, existe um consenso sobre as áreas que cada um deles abrange (veja, por exemplo, Vranesic, 1980, Siewiorek, 1982, e Bell e outros, 1978a).

O termo 'arquitetura de um computador' refere-se aos atributos de um sistema que são visíveis para o programador ou, em outras palavras, aos atributos que têm impacto direto sobre a execução lógica de um programa. O termo 'organização de um computador' refere-se às unidades operacionais e suas interconexões que implementam as especificações da sua arquitetura. Exemplos de atributos de arquitetura incluem o conjunto de instruções, o número de bits usados para representar os vários tipos de dados (por exemplo, números, caracteres), os mecanismos de E/S e as técnicas de endereçamento à memória. Atributos de organização incluem detalhes de hardware transparentes ao programador, tais como os sinais de controle, as interfaces entre o computador e os periféricos e a tecnologia de memória utilizada.

Definir se um computador deve ou não ter uma instrução de multiplicação, por exemplo, constitui uma decisão do projeto da sua arquitetura. Por outro lado, definir se essa instrução será implementada por uma unidade especial de multiplicação ou por um mecanismo que utiliza repetidamente sua unidade de soma constitui uma decisão do projeto da sua organização. Essa decisão de organização pode ser baseada na previsão sobre a freqüência de uso da instrução de multiplicação, na velocidade relativa das duas abordagens e no custo e tamanho físico da unidade especial de multiplicação.

Historicamente, e ainda hoje, a distinção entre arquitetura e organização é de fundamental importância. Muitos fabricantes de computador oferecem uma família de modelos de computadores, todos com a mesma arquitetura, mas com diferenças de organização. Dessa

maneira, os diferentes modelos da família têm preços e características de desempenho distintos. Além disso, uma arquitetura pode sobreviver por muitos anos, enquanto sua organização muda com a evolução da tecnologia. Um exemplo claro desses dois fenômenos é o da arquitetura do Sistema 370 da IBM. Essa arquitetura foi introduzida em 1970 e com um grande número de modelos. Um cliente com exigências mais modestas podia comprar um modelo mais barato e mais lento e, caso sua demanda por desempenho aumentasse, ele poderia migrar para um modelo mais rápido e mais caro, sem ter de abandonar as aplicações que já tivessem sido desenvolvidas. Ao longo dos anos, a IBM introduziu muitos modelos novos, com tecnologia aprimorada, para substituir os modelos mais antigos, oferecendo ao cliente maior velocidade, menor custo ou ambos. Esses modelos mais novos conservavam a mesma arquitetura, preservando o investimento em software do cliente. Notavelmente, a arquitetura do Sistema 370 sobreviveu até hoje, com pequenos melhoramentos, como a arquitetura da linha de computadores de grande porte da IBM.

Na classe de sistemas denominados microcomputadores, a relação entre arquitetura e organização é muito mais estreita. Mudanças na tecnologia não apenas influenciam a organização, mas também resultam na introdução de arquiteturas mais ricas e poderosas. Para essas máquinas menores, geralmente não existe um forte requisito de compatibilidade de uma geração para outra. Portanto, no caso dessas máquinas, existe maior relação entre as decisões relativas à sua arquitetura e à sua organização. Um exemplo intrigante são os computadores com um conjunto reduzido de instruções (*reduced instruction set computer* — RISC), que examinamos no Capítulo 12.

Conforme salientado, este livro aborda tanto a arquitetura quanto a organização de computadores, sendo talvez maior a ênfase com relação à organização. Entretanto, como a organização deve ser projetada para implementar uma especificação particular de arquitetura, um tratamento minucioso da organização de computadores requer também um exame detalhado de sua arquitetura.

1.2 ESTRUTURA E FUNÇÃO

Um computador é um sistema de grande complexidade; computadores modernos contêm milhões de componentes eletrônicos elementares. Como é possível, então, descrevê-los com clareza? O ponto-chave é o reconhecimento da natureza hierárquica da maioria dos sistemas complexos, incluindo o computador (Simon, 1969). Um sistema hierárquico é constituído de um conjunto de subsistemas inter-relacionados, cada qual, por sua vez, possuindo também uma estrutura hierárquica, contendo, em seu nível mais baixo, subsistemas elementares.

A natureza hierárquica dos sistemas complexos é essencial tanto para seu projeto quanto para sua descrição. Em cada momento, o projetista precisa lidar apenas com um nível particular do sistema. Em cada nível, o sistema consiste em um conjunto de componentes e de relacionamentos entre estes. O comportamento de cada nível depende apenas de uma caracterização abstrata e simplificada do sistema de nível imediatamente inferior. O projetista deve considerar, em cada nível, sua estrutura e o funcionamento dos seus componentes:

- **Estrutura:** o modo como os componentes estão inter-relacionados.
- **Função:** a operação de cada componente individual como parte da estrutura.

Existem duas opções possíveis para a descrição desses sistemas: começando do nível mais baixo e compondo as partes até a obtenção de uma descrição mais global ou começando com uma visão do nível mais alto e decompondo o sistema em suas subpartes. A experiência com a descrição de sistemas dessa natureza, em diversas áreas, sugere que a abordagem de cima para baixo é mais clara e eficaz (Weinberg, 1975).

A abordagem adotada neste livro segue este ponto de vista: os sistemas de computação são descritos a partir do nível mais alto para o mais baixo. Primeiramente, abordamos os componentes do nível mais alto do sistema, descrevendo sua estrutura e suas funções, e prosseguimos, sucessivamente, para as camadas inferiores da hierarquia. O restante desta seção fornece uma visão sucinta dessa abordagem.

Função

Tanto a estrutura quanto as funções de um computador são, em sua essência, muito simples. A Figura 1.1 representa as funções básicas que um computador pode desempenhar. Em termos gerais, existem apenas quatro:

- Processamento de dados
- Armazenamento de dados
- Transferência de dados
- Controle

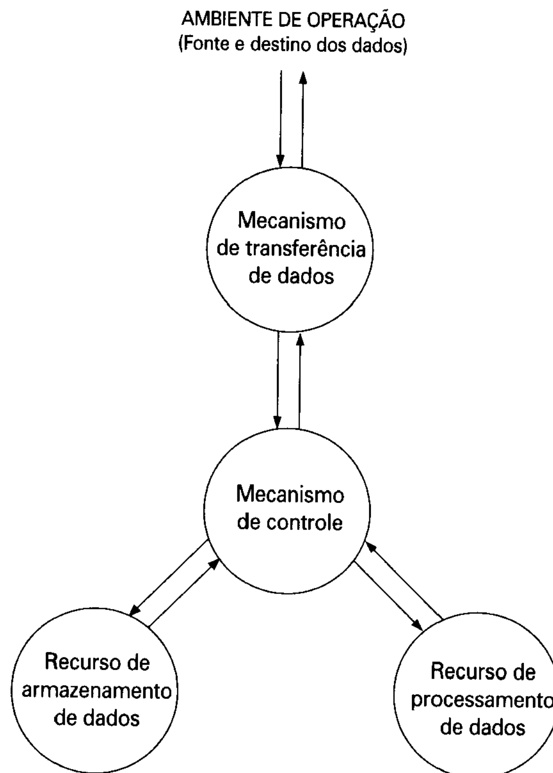


Figura 1.1 Visão funcional de um computador.

É claro que um computador deve ser capaz de *processar dados*. Os dados podem ter grande variedade de tipos, e a gama de requisitos de processamento é muito ampla. Entretanto, veremos que existem poucos métodos ou tipos fundamentais de processamento de dados.

É também essencial que um computador *armazene dados*. Mesmo quando é realizado um processamento de dados do tipo *on the fly* (isto é, quando os dados de entrada são processados e os resultados são enviados diretamente para a saída), o computador precisa armazenar temporariamente ao menos aquela porção dos dados que está sendo processada naquele instante. Portanto, existe pelo menos uma função de armazenamento temporário de dados. É igualmente importante que um computador seja capaz de armazenar dados de maneira permanente, por períodos mais longos. Os dados são armazenados no computador, para subseqüente recuperação e modificação.

Um computador deve ser capaz de *transferir dados*, tanto internamente quanto com o mundo externo. O ambiente de operação de um computador consiste em dispositivos que servem como fonte ou como destino de dados. Quando os dados são recebidos ou enviados para um dispositivo diretamente conectado ao computador, o processo é conhecido como *entrada e saída* (E/S) e o dispositivo é denominado um *periférico*. Quando os dados são transferidos por distâncias maiores, de ou para um dispositivo remoto, o processo é conhecido como *comunicação de dados*.

Finalmente, deve existir um *controle* dessas três funções. Em última instância, esse controle é exercido pelo(s) indivíduo(s) que fornece(m) instruções ao computador. Dentro de um sistema de computação, uma unidade de controle gerencia os recursos do computador e rege o desempenho das suas partes funcionais em resposta a essas instruções.

Nesse nível genérico de discussão, o número de operações possíveis que podem ser desempenhadas é pequeno. A Figura 1.2 representa os quatro tipos de operações possíveis. O computador pode funcionar simplesmente como um dispositivo de transferência de dados de um periférico ou de uma linha de comunicação para outro (Figura 1.2a). Pode também funcionar como um dispositivo de armazenamento de dados (Figura 1.2b), sendo os dados transferidos do ambiente externo para a memória do computador (leitura) e vice-versa (escrita). Os dois últimos diagramas mostram operações envolvendo processamento de dados, seja sobre dados armazenados na memória (Figura 1.2c), seja sobre dados transferidos entre a memória e o ambiente externo (Figura 1.2d).

Essa discussão pode parecer absurdamente genérica. Com certeza é possível diferenciar, mesmo no nível mais alto da estrutura de um computador, uma grande variedade de funções. Entretanto, de acordo com Siewiorek e outros (1982), a estrutura de um computador notoriamente não reflete a função que ele desempenha. Isso se deve, principalmente, à sua natureza de dispositivo de propósito geral, em que qualquer especialização funcional decorre da sua programação e não do seu projeto.

Estrutura

A Figura 1.3 constitui a representação mais simples possível de um computador. O computador é uma entidade que interage, de alguma maneira, com seu ambiente externo. Em geral, todas as suas ligações com o ambiente externo podem ser classificadas como dispositivos periféricos ou como linhas de comunicação. Esses dois tipos de ligação serão abordados no decorrer deste capítulo.

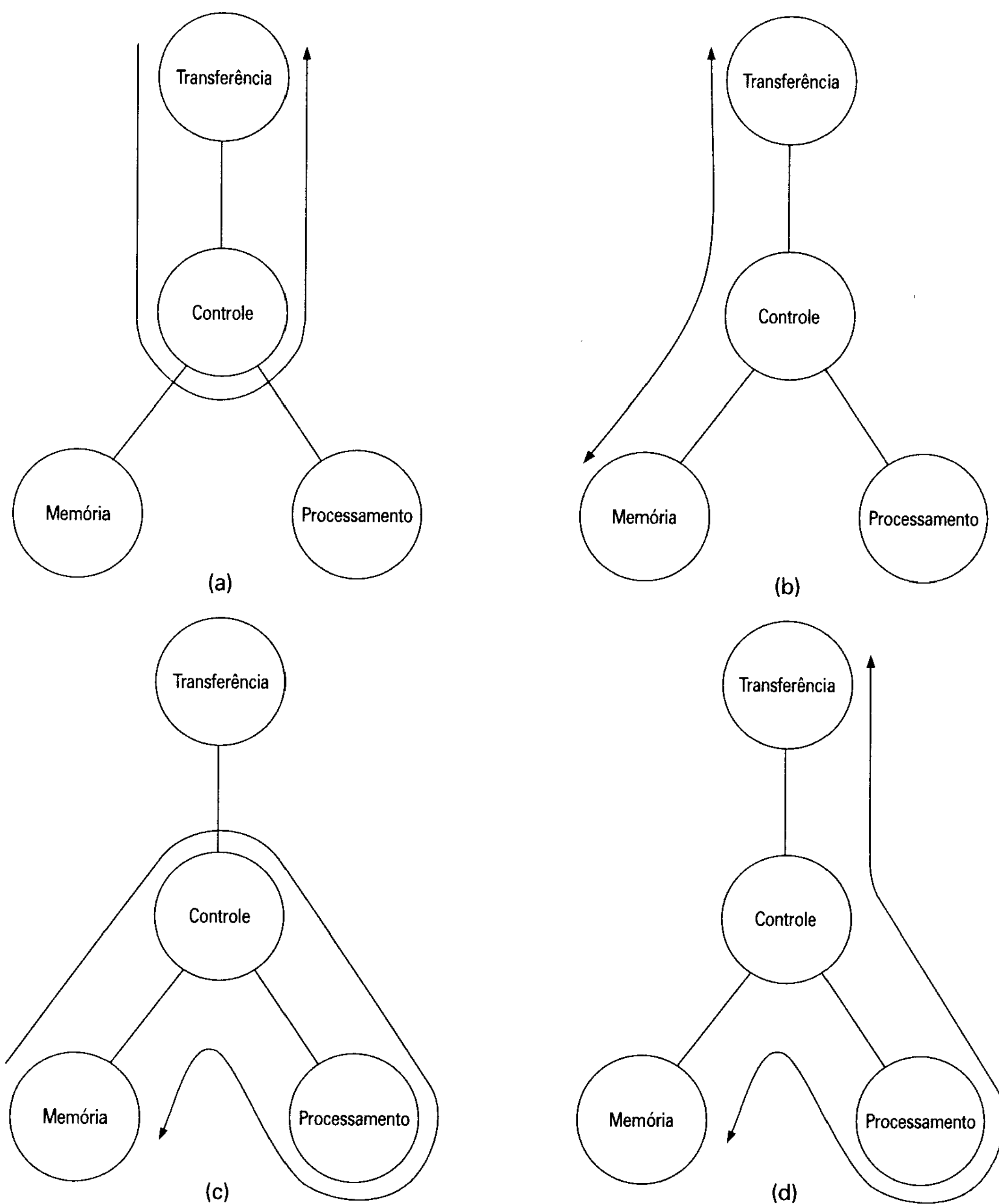


Figura 1.2 Operações possíveis em um computador.

Nosso maior interesse, neste livro, reside na estrutura interna do próprio computador, que é mostrada, em um nível mais alto, na Figura 1.4. Há quatro principais componentes estruturais:

- **Unidade central de processamento (CPU):** controla a operação do computador e desempenha funções de processamento de dados. É muitas vezes chamada, simplesmente, de *processador*.
- **Memória principal:** armazena dados.
- **E/S:** transfere dados entre o computador e o ambiente externo.
- **Sistema de interconexão:** mecanismos que estabelecem a comunicação entre a CPU, a memória principal e os dispositivos de E/S (entrada/saída).

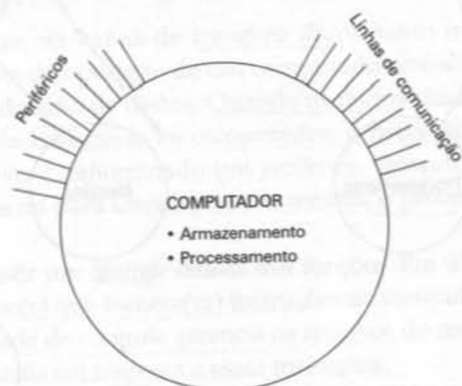


Figura 1.3 O computador.



Figura 1.4 O computador: estrutura de alto nível.

Um sistema de computação pode ter um ou mais de cada um dos componentes relacionados anteriormente. Os sistemas tradicionais são compostos de uma única CPU. Nos últimos anos, tem havido um crescente uso de sistemas com vários processadores. Algumas questões relativas ao projeto de sistemas com vários processadores são discutidas no decorrer deste texto; o Capítulo 16 aborda esses sistemas.

Cada um dos componentes de um computador é examinado, detalhadamente, na Parte 2. Entretanto, o componente de nosso maior interesse e, de certa maneira, também o mais complexo é a CPU; sua estrutura é representada na Figura 1.5 e seus principais componentes estruturais são os seguintes:

- **Unidade de controle:** controla a operação da CPU e, portanto, do computador.
- **Unidade lógica e aritmética (ULA):** desempenha as funções de processamento de dados do computador.
- **Registradores:** fornecem o armazenamento interno de dados para a CPU.
- **Interconexão da CPU:** mecanismo que possibilita a comunicação entre a unidade de controle, a ULA e os registradores.

Cada um desses componentes é examinado detalhadamente na Parte 3, na qual veremos que o uso de técnicas de paralelismo e de *pipelines* aumenta ainda mais a complexidade. Finalmente, existem diversas abordagens para a implementação da unidade de controle, sendo *microprogramação* a mais comum. Nessa abordagem, a estrutura da unidade de controle pode ser representada como na Figura 1.6. Essa estrutura é examinada na Parte 4.

1.3 ESTRUTURA DO LIVRO

Este capítulo serve como uma introdução para o restante do livro. Um breve resumo de cada um dos demais capítulos é apresentado a seguir.

Evolução e desempenho de computadores

O Capítulo 2 atende a dois propósitos. O primeiro consiste em introduzir os conceitos básicos de arquitetura e organização de computadores, o que se faz, de modo que torne a leitura mais fácil e interessante, por meio de uma descrição da evolução histórica da tecnologia de computadores. Esse capítulo trata também das tendências tecnológicas que colocaram o desempenho como o foco principal dos projetos de sistemas de computação, assim como apresenta várias técnicas e estratégias usadas para se obter um desempenho balanceado e eficiente.

Barramentos do sistema

No nível mais alto, um computador é constituído de um processador, de uma memória e de componentes de E/S. O comportamento funcional do sistema consiste na troca de dados e de sinais de controle entre esses componentes. Para possibilitar essa transferência de dados e de sinais de controle, os componentes devem ser interconectados. O Capítulo 3 começa com uma breve descrição dos componentes de um computador e dos seus requisitos de entrada e saída. Em seguida, aborda os principais aspectos que afetam o projeto do sistema de interconexão, em particular a necessidade de fornecer suporte a interrupções. A maior parte desse capítulo é dedicada ao estudo da abordagem mais utilizada para o sistema de interconexão: o uso de uma estrutura de barramentos.

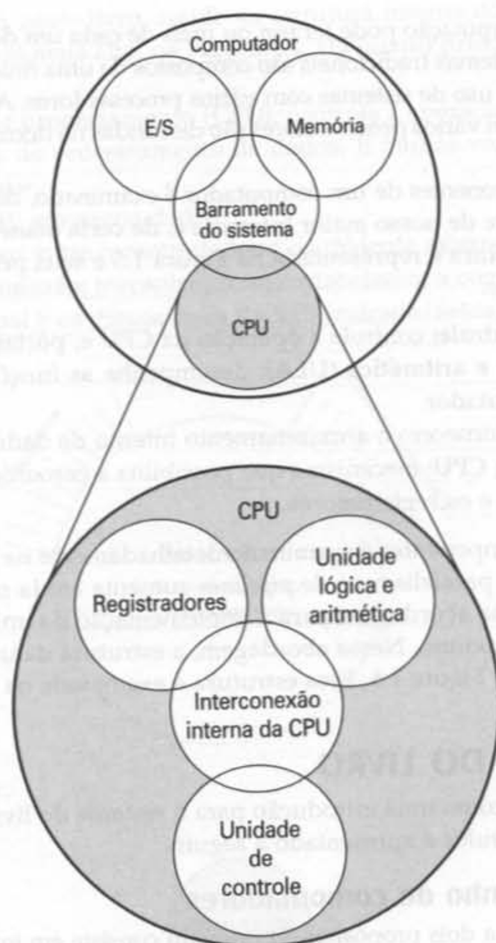


Figura 1.5 A unidade central de processamento (CPU).

Memória interna

A memória de um computador apresenta grande diversidade em relação ao tipo, à tecnologia, à organização, ao desempenho e ao custo. Um sistema de computação típico é equipado com uma hierarquia de subsistemas de memória, sendo algumas delas internas (diretamente acessíveis pelo processador) e outras externas (acessíveis pelo processador por meio de um módulo de E/S). O Capítulo 4 inicia com uma visão geral dessa hierarquia de memórias e então enfatiza os aspectos de projeto relacionados à memória interna. Primeiramente, descrevem-se a natureza e a organização de uma memória principal de semicondutor. Em seguida, trata-se detalhadamente do projeto de memórias cache, incluindo memórias cache para código e para dados, e memórias cache em dois níveis. Finalmente, são abordadas as organizações de memória DRAM (*dynamic random access-memory* — memória dinâmica de acesso aleatório) mais avançadas.

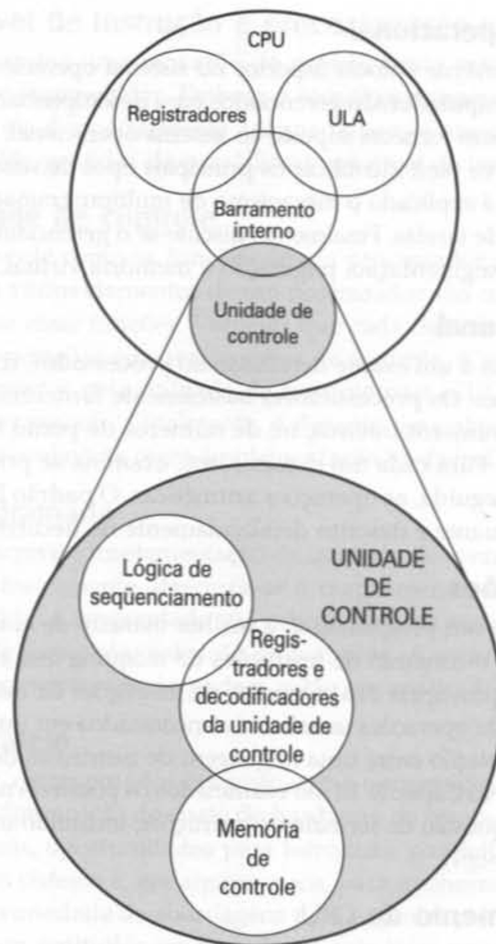


Figura 1.6 A unidade de controle.

Memória externa

O Capítulo 5 aborda diversos parâmetros de projeto e de desempenho de memórias de disco. Aqui são examinados os esquemas RAID (*redundant array* of *independent disks* — agrupamento redundante de discos independentes), que têm se tornado cada vez mais comuns. Além disso, são abordados os sistemas de memória óptica e de fita magnética.

Entrada e saída

Os módulos de E/S são interconectados ao processador e à memória principal, e cada um controla um ou mais dispositivos externos. O Capítulo 6 examina o mecanismo de interação entre os módulos de E/S e o restante do sistema de computação, por meio das técnicas de E/S programada, E/S por interrupção e acesso direto à memória (DMA — *direct memory access*). Também é descrita a interface entre um módulo de E/S e os dispositivos externos.

Suporte ao sistema operacional

Nesse ponto, é conveniente enfocar aspectos do sistema operacional, explicando como os componentes básicos do computador são gerenciados para desempenhar um trabalho útil e como o hardware é organizado para fornecer suporte ao sistema operacional. O Capítulo 7 inicia com um breve histórico, que serve para identificar os principais tipos de sistemas operacionais e motivar seu uso. Em seguida, é explicado o mecanismo de multiprogramação, com a descrição das funções de escalonamento de tarefas. Finalmente, discute-se o gerenciamento da memória, abordando os mecanismos de segmentação, paginação e memória virtual.

Aritmética computacional

O Capítulo 8 dá início a um exame detalhado do processador, com uma discussão sobre aritmética de computadores. Os processadores basicamente fornecem suporte a dois tipos de aritmética: aritmética de números inteiros, ou de números de ponto fixo, e aritmética de números de ponto flutuante. Para cada um desses casos, examina-se primeiramente a representação dos números e, em seguida, as operações aritméticas. O padrão IEEE 754 para aritmética de números de ponto flutuante é descrito detalhadamente no decorrer do capítulo.

Conjuntos de instruções

Do ponto de vista de um programador, a melhor maneira de entender a operação de um processador é conhecendo o conjunto de instruções de máquina que ele executa. O Capítulo 9 examina as características principais dos conjuntos de instruções de máquina e aborda diversos tipos de dados e os tipos de operações usualmente encontrados em um conjunto de instruções. Em seguida, explica-se a relação entre uma linguagem de instruções de um processador e uma linguagem de montagem. No Capítulo 10 são examinados os possíveis modos de endereçamento. Finalmente, é abordada a questão do formato de instruções, incluindo uma discussão sobre compromissos de projeto (*trade-offs*).

Estrutura e funcionamento da CPU

O Capítulo 11 é dedicado a uma discussão sobre a estrutura interna e o funcionamento do processador. Primeiramente, revê-se a organização global de um processador (ULA, unidade de controle, conjunto de registradores). Em seguida, discute-se a organização do seu conjunto de registradores. O restante do capítulo descreve o funcionamento de um processador durante a execução das instruções de máquina. O ciclo de execução de uma instrução é descrito, mostrando-se o funcionamento e o inter-relacionamento entre os ciclos de busca, de endereçamento indireto, de execução e de interrupção. Finalmente, discute-se detalhadamente o uso de *pipelines* para se obter melhor desempenho.

Computadores RISC

Uma das mais significativas inovações na arquitetura e organização de computadores nos últimos anos se deu com a arquitetura de computadores com um conjunto reduzido de instruções (RISC). A arquitetura RISC constitui um desvio dramático da tendência histórica verificada na arquitetura de processadores. Uma análise dessa abordagem traz à luz muitas questões importantes relativas à arquitetura e à organização de computadores. O Capítulo 12 descreve a abordagem RISC, comparando-a com a abordagem CISC (*complex instruction set computer* — computador com um conjunto complexo de instruções).

Paralelismo em nível de instrução e processadores superescalares

O Capítulo 13 examina uma inovação de projeto ainda mais recente e igualmente importante: o processador superescalar. Embora a tecnologia superescalar possa ser usada em qualquer processador, ela é especialmente adequada para a arquitetura RISC. Esse capítulo aborda também a questão genérica de paralelismo em nível de instrução.

Operação da unidade de controle

O Capítulo 14 discute como as funções de um processador são realizadas ou, mais especificamente, como os vários elementos de um processador são controlados pela unidade de controle para apresentar essas funções. Veremos que cada ciclo de instrução é constituído de um conjunto de microoperações que geram sinais de controle. A execução é completada pelo efeito desses sinais, enviados pela unidade de controle para a ULA, para os registradores e para a estrutura de interconexão. Finalmente, é descrita uma abordagem de implementação da unidade de controle conhecida como implementação *hardwired*.

Controle microprogramado

O Capítulo 15 descreve a implementação da unidade de controle utilizando a técnica de microprogramação. Primeiramente, descreve-se o mapeamento das microoperações em microinstruções. Em seguida, é apresentado um esboço de uma memória de controle, que contém um microprograma para cada instrução de máquina. A estrutura e o funcionamento da unidade de controle microprogramada podem então ser explicados.

Processamento paralelo

Tradicionalmente, o computador era visto como uma máquina seqüencial. Com a evolução da tecnologia e a diminuição do custo do hardware de computadores, os projetistas têm procurado, cada vez mais, oportunidades para introduzir paralelismo, geralmente para melhorar o desempenho do sistema e, em alguns casos, para melhorar sua confiabilidade. O Capítulo 16 examina uma variedade de abordagens para organização paralela de computadores. No caso de sistemas com múltiplos processadores, este livro apresenta também um estudo das questões de projeto relativo ao problema de coerência das memórias cache.

Lógica digital

Este livro aborda os elementos da memória binária e as funções digitais como blocos fundamentais na construção de sistemas de computação. Este apêndice descreve como esses elementos de memória e essas funções podem ser implementados em lógica digital. Começa com uma breve revisão de álgebra booleana e, em seguida, é introduzido o conceito de porta lógica. Finalmente, discutem-se os circuitos combinatórios e seqüenciais, que podem ser construídos a partir de portas lógicas.

1.4 INTERNET E RECURSOS NA WEB

Está disponível na Internet uma variedade de informações para apoio ao ensino de arquitetura e organização de computadores por meio deste livro, assim como diversas informações sobre recentes pesquisas e evoluções na área.

Sites Web deste livro



Consulte o Companion Website (CW) deste livro no endereço www.prenhall.com/stallings_br. Ali você encontra recursos adicionais para o professor, além de um link para o site do autor, onde estão disponíveis diversos outros recursos para alunos e professores. Veja na página XI uma descrição detalhada desse site.

Outros sites Web

Existe uma variedade de sites Web que fornecem informações relacionadas aos tópicos tratados neste livro. Indicações sobre sites Web específicos são apresentadas nos capítulos subseqüentes, na seção "Leitura e sites Web recomendados". Como os endereços de sites Web tendem a ser freqüentemente alterados, os endereços desses sites não foram incluídos no livro. O endereço apropriado de cada site citado pode ser encontrado no site Web deste livro.

Os seguintes sites Web contêm informações de interesse geral sobre arquitetura e organização de computadores:

- **WWW Computer Architecture Home Page:** um índice abrangente de informações relevantes sobre pesquisas em arquitetura de computadores, incluindo grupos e projetos de pesquisa, organizações tecnológicas, literatura, anúncios de empregos e informações comerciais.
- **CPU Info Center:** informações sobre processadores específicos, incluindo documentação técnica, informações sobre produtos e anúncios mais recentes.
- **ACM Special Interest Group on Computer Architecture:** informações sobre atividades e publicações do SIGARCH.
- **IEEE Technical Commitee on Computer Architecture:** cópias atualizadas do periódico TCAA.
- **Intel Technology Journal:** publicações on-line da Intel.

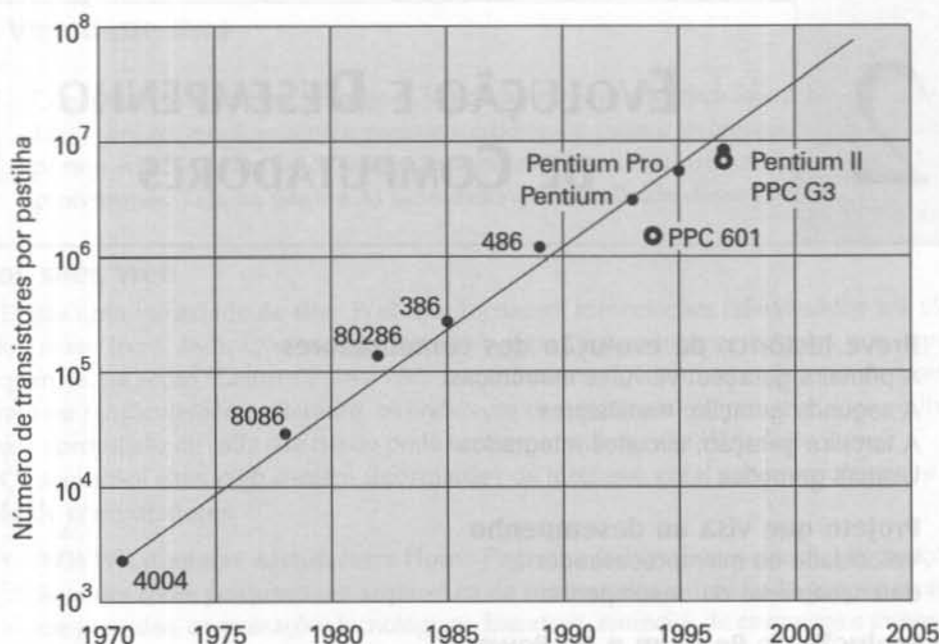
Grupos de notícias USENET

Uma variedade de grupos USENET é dedicada a aspectos de arquitetura e organização de computadores. Assim como na maioria dos grupos USENET, existe uma alta taxa de ruído, mas vale a pena experimentar alguns deles para verificar se incluem discussões do seu interesse. Os mais relevantes são:

- **comp.arch:** grupo de discussão genérica sobre arquitetura de computadores. Em geral muito bom.
- **comp.arch.arithmetic:** discute padrões e algoritmos de aritmética de computadores.
- **comp.arch.storage:** a discussão abrange desde produtos e tecnologia até questões de caráter mais prático.

capítulo

2 Evolução e Desempenho de Computadores



- A evolução dos computadores tem sido caracterizada pelo aumento da velocidade dos processadores, pela diminuição do tamanho dos componentes, pelo aumento da capacidade da memória e pelo aumento da capacidade e da velocidade de E/S.
- Um dos fatores responsáveis pelo grande aumento da velocidade dos processadores é a diminuição do tamanho dos componentes dos microprocessadores; isso acarreta a redução da distância entre os componentes e, conseqüentemente, o aumento da velocidade. Entretanto, os ganhos reais de velocidade obtidos nos últimos anos são devidos principalmente a mudanças na organização do processador, incluindo o uso intensivo de *pipeline* e de técnicas de execução paralela de instruções, assim como de técnicas de execução especulativa, que consistem na tentativa de executar, antecipadamente, instruções que possam vir a ser requeridas. Todas essas técnicas são projetadas a fim de manter o processador ocupado o maior tempo possível.
- Um aspecto crítico no projeto de sistemas de computação é o balanceamento do desempenho dos diversos elementos, para que o ganho de desempenho obtido em uma área não seja prejudicado por um atraso em outra área. Em particular, a velocidade do processador tem aumentado muito mais rapidamente do que a velocidade de acesso à memória. Várias técnicas são empregadas para compensar esse desequilíbrio, incluindo memórias cache, vias de comunicação de dados de maior largura entre a memória e o processador e pastilhas de memória mais inteligentes.

Começamos nosso estudo com um breve histórico da evolução dos computadores. Esse histórico, além de interessante, fornece uma visão da estrutura e das funções de um computador. Em seguida, abordamos a questão do desempenho. A justificativa aí apresentada para a necessidade de balancear a utilização dos diversos recursos de um computador aborda um contexto que nos será útil ao longo do livro. Finalmente, examinamos brevemente a evolução dos dois sistemas que servem como exemplos básicos em todo o livro: o Pentium e o PowerPC.

2.1 BREVE HISTÓRICO DA EVOLUÇÃO DOS COMPUTADORES

A primeira geração: válvulas eletrônicas

ENIAC

O ENIAC (Computador e Integrador Numérico Eletrônico — *Electronic Numerical Integrator and Computer*), projetado e construído sob a supervisão de John Mauchly e John Presper Eckert na Universidade da Pensilvânia, foi o primeiro computador eletrônico digital de propósito geral em todo o mundo.

O projeto foi uma resposta às necessidades dos Estados Unidos diante da guerra. O Laboratório de Pesquisas Balísticas do Exército americano (Army's Ballistics Research Laboratory — BRL), órgão responsável por desenvolver tabelas de trajetória e alcance para as novas armas, vinha encontrando dificuldades em obter essas tabelas com boa precisão e em tempo hábil. Sem elas, as novas armas de artilharia seriam inúteis. O BRL empregava mais de 200 pessoas que, utilizando calculadoras de mesa, resolviam as equações de balística necessárias. A preparação das tabelas para uma única arma consumia várias horas de trabalho de uma pessoa, até mesmo dias.

Mauchly, um professor de engenharia elétrica da Universidade da Pensilvânia, e Eckert, um de seus alunos de pós-graduação, propuseram a construção de um computador de propósito geral para as aplicações do BRL, utilizando válvulas. Em 1943, a proposta foi aceita pelo Exército americano e o trabalho no ENIAC teve início. O resultado foi uma máquina enorme que pesava 30 toneladas, ocupava espaço de aproximadamente 140 metros quadrados e continha mais de 18 mil válvulas. A operação dessa máquina consumia 140 quilowatts de energia elétrica. Ela era muito mais rápida do que qualquer computador eletromecânico, sendo capaz de executar 5 mil adições por segundo.

O ENIAC era uma máquina decimal e não uma máquina binária; ou seja, a representação dos números era feita na base decimal, a qual era utilizada também para a realização das operações aritméticas. A memória consistia em 20 'acumuladores', cada um dos quais capaz de armazenar um número decimal de dez dígitos. Cada dígito era representado por um anel de dez válvulas. A cada instante, apenas uma válvula ficava no estado ON (ligado), representando um dos dez dígitos. A principal desvantagem do ENIAC era que ele tinha de ser programado manualmente, ligando e desligando chaves e conectando e desconectando cabos.

O ENIAC foi concluído em 1946, tarde demais para ser utilizado durante a guerra. Sua primeira tarefa foi realizar uma série de cálculos complexos, empregados para ajudar a determinar se a bomba H poderia ser construída. O emprego do ENIAC para um propósito diferente daquele para o qual fora originariamente projetado demonstrou seu caráter de computador de propósito geral. O ENIAC permaneceu operando no BRL até 1955, quando foi desativado.

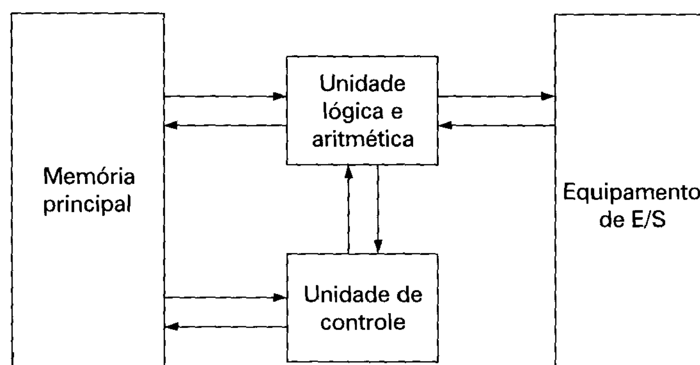


Figura 2.1 Estrutura do computador IAS.

A máquina de von Neumann

A tarefa de carregar e de modificar um programa no ENIAC era extremamente tediosa. O processo de programação poderia ser extremamente facilitado se um programa pudesse ser representado de maneira adequada, de modo que fosse armazenado na memória, juntamente com os dados. Assim, o computador poderia obter as instruções diretamente, a partir da memória, e um programa poderia ser carregado ou modificado simplesmente atribuindo valores a posições de memória.

Essa idéia, conhecida como *conceito de programa armazenado*, geralmente é atribuída aos projetistas do ENIAC, principalmente ao matemático John von Neumann, que era um dos consultores no projeto do ENIAC. Ela foi simultaneamente concebida por Alan Turing. A primeira publicação da idéia, concretizada em uma proposta formulada por von Neumann, ocorreu em 1945, para um novo computador, o EDVAC (Computador Variável Discreto Eletrônico — *Electronic Discrete Variable Computer*).

Em 1946, von Neumann e seus colegas começaram o projeto de um novo computador de programa armazenado, conhecido como IAS, no Instituto de Estudos Avançados de Princeton. O IAS, embora concluído somente em 1952, constitui o protótipo de todos os computadores de propósito geral subseqüentes.

A Figura 2.1 mostra a estrutura geral do IAS, que consiste em:

- Uma memória principal, que armazena dados e instruções.
- Uma unidade lógica e aritmética (ULA), capaz de realizar operações com dados binários.
- Uma unidade de controle, que interpreta e executa instruções armazenadas na memória.
- Dispositivos de entrada e saída (E/S), operados pela unidade de controle.

Esta estrutura foi esboçada em uma proposta anterior de von Neumann, a qual vale a pena reproduzir aqui (von Neumann, 1945):

> 2.2 **Primeiro:** como o dispositivo é, em essência, um computador, deverá executar, mais freqüentemente, as operações elementares da aritmética: adição, subtração, multiplicação e divisão. É razoável, portanto, que deva conter componentes especializados para realizar essas operações.

Deve-se observar, entretanto, que, embora esse princípio seja provavelmente correto, a maneira como será implementado requer um estudo meticuloso... De qualquer modo, deverá existir, provavelmente, uma *unidade central de aritmética,* que constituirá *a primeira parte específica do dispositivo: CA.*

2.3 **Segundo:** o controle lógico do dispositivo, ou seja, a execução das operações na seqüência apropriada, pode ser feito, de modo mais eficiente, por meio de um componente de controle central. Se o dispositivo tiver de ser *flexível,* isto é, se tiver de ser um dispositivo de *propósito geral,* será conveniente, tanto quanto possível, distinguir o conjunto de instruções específicas para a solução de um determinado problema e os componentes de controle geral que se encarregam da execução dessas instruções, independentemente de quais elas sejam. As instruções devem ser armazenadas de algum modo; os componentes de controle são descritos pelas partes operacionais definidas do dispositivo. Entendemos como *controle central* apenas essa última função, e os componentes que a desempenham constituem *a segunda parte específica do dispositivo: CC.*

2.4 **Terceiro:** qualquer dispositivo destinado à execução de longas e complicadas seqüências de operações (especificamente de cálculos) deve ter uma memória considerável....

(b) O conjunto de instruções para a solução de um problema complicado pode ter tamanho considerável, particularmente se o código for circunstancial (o que ocorre na maioria dos casos). Esse conjunto de instruções deve ser, de alguma maneira, recuperado....

A *memória,* como um todo, constitui *a terceira parte específica do dispositivo: M.*

2.6 As três partes específicas, CA, CC e M, correspondem aos neurônios *associativos* do sistema nervoso humano. Resta discutir os componentes equivalentes aos neurônios *sensoriais,* ou *aferentes,* e aos neurônios *motores,* ou *eferentes.* Esses são os elementos de *entrada* e *saída* do dispositivo...

O dispositivo deve ser dotado da habilidade de manter contato de entrada e saída (sensorial e motor) com alguns mecanismos específicos dessa natureza. Estes mecanismos serão denominados *meios de armazenamento externo do dispositivo: A...*

2.7 **Quarto:** o dispositivo deve possuir elementos para transferir... informações de A para seus componentes específicos C e M. Esses elementos constituem sua *entrada, a quarta parte específica do dispositivo: E.* Veremos que é mais adequado efetuar todas as transferências de A (por E) para M e nunca diretamente de C...

2.8 **Quinto:** o dispositivo deve possuir elementos para transferir... de seus componentes específicos C e M para A. Esses elementos constituem sua *saída, a quinta parte específica do dispositivo: S.* Veremos que novamente é mais adequado efetuar todas as transferências de M (por S) para A e nunca diretamente de C.

Com raras exceções, todos os computadores atuais possuem essas mesmas funções e estrutura geral e assim são conhecidos como máquinas com arquitetura de von Neumann. Por isso, é apropriado incluir, nesse ponto, uma breve descrição da operação do computador IAS (Burks, 1946). De acordo com Hayes (1988), a terminologia e a notação de von Neumann são substituídas, a seguir, por termos correspondentes usados hoje em dia; os exemplos e as ilustrações que acompanham esta discussão são baseados no texto anterior.

A memória do IAS consiste em mil posições de memória, denominadas *palavras*, cada uma constituída de 40 dígitos binários (bits). Dados e instruções são ambos armazenados na memória. Portanto, os números devem ser representados em forma binária e cada instrução deve ter também um código binário. A Figura 2.2 ilustra estes formatos. Cada número é representado por um bit de sinal e um valor de 39 bits. Uma palavra pode conter duas instruções de 20 bits, cada uma consistindo em um código de operação (*opcode*) com 8 bits, que especifica a operação a ser executada, e de um endereço com 12 bits, que designa uma palavra na memória (numerada de 0 a 999).

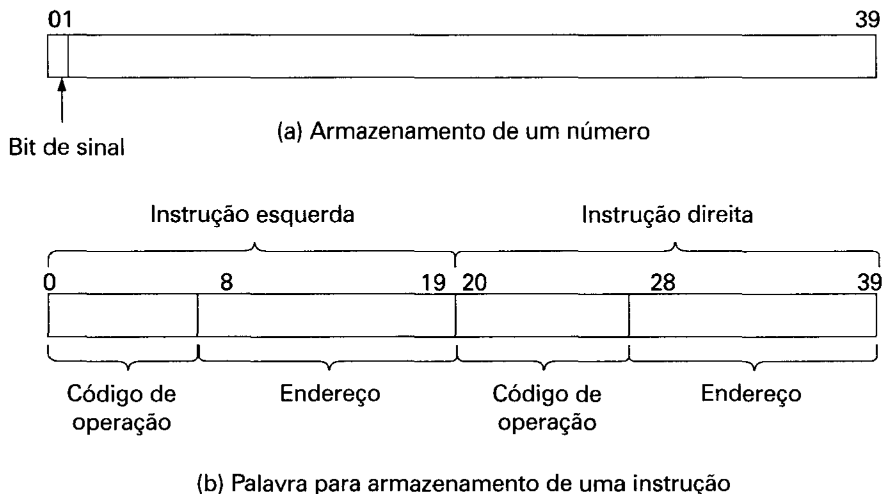


Figura 2.2 Formatos de uma palavra no IAS.

A unidade de controle controla a operação do IAS, efetuando a busca das instruções na memória e executando-as, uma de cada vez. Para entender essa operação, é necessário um diagrama de estrutura mais detalhado, como indicado na Figura 2.3. Essa figura revela que tanto a unidade de controle quanto a ULA contêm células de armazenamento denominadas *registradores*, classificados como segue:

- **Registrador temporário de dados (*Memory Buffer Register* — MBR):** contém uma palavra com dados a ser armazenada na memória ou é utilizado para receber uma palavra da memória.
- **Registrador de endereçamento à memória (*Memory Address Register* — MAR):** especifica o endereço, na memória, da palavra a ser escrita ou lida no MBR.
- **Registrador de instruções (*Instruction Register* — IR):** contém o código de operação de 8 bits que está sendo executado.
- **Registrador de armazenamento temporário de instruções (*Instruction Buffer Register* — IBR):** é utilizado para armazenar temporariamente a instrução contida na porção à direita de uma palavra da memória.
- **Contador do programa (*Program Counter* — PC):** contém o endereço de memória do próximo par de instruções a ser buscado da memória.

- **Acumulador (*Accumulator* — AC) e Quociente de Multiplicação (*Multiplier Quotient* — MQ):** são utilizados para armazenar temporariamente os operandos e o resultado de operações efetuadas na ULA. Por exemplo, o resultado da multiplicação de dois números de 40 bits é um número de 80 bits; os 40 bits mais significativos são armazenados no acumulador (AC) e os 40 bits menos significativos, no registrador de quociente de multiplicação (MQ).

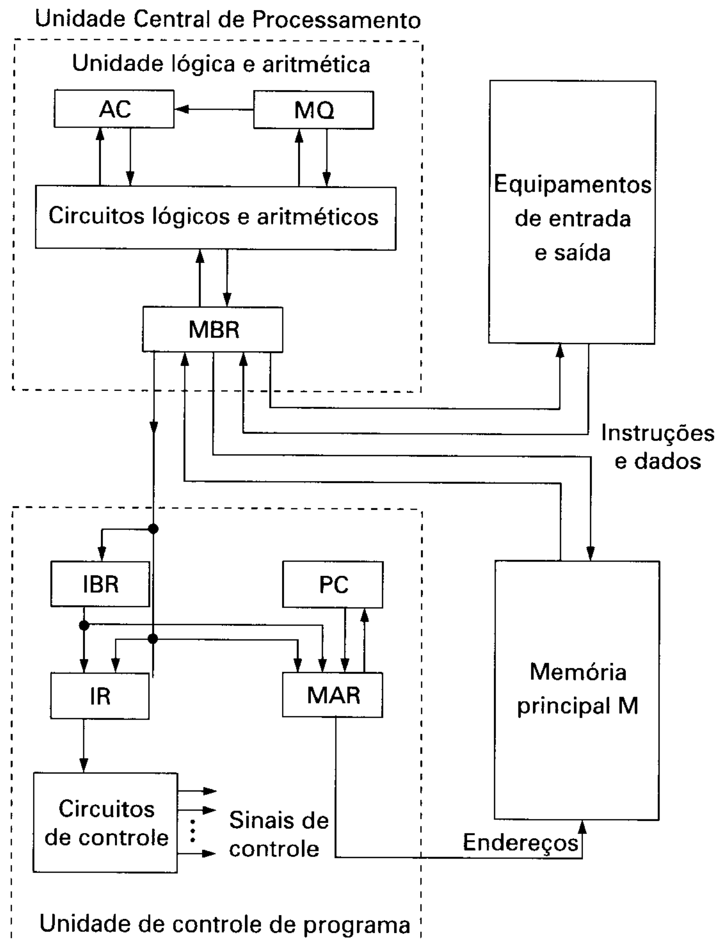


Figura 2.3 Estrutura detalhada do IAS.

A operação do IAS consiste na execução repetida de um *ciclo de instruções*, como mostrado na Figura 2.4. Cada ciclo de instruções consiste em dois subciclos. Durante o *ciclo de busca*, o código de operação da próxima instrução é carregado no IR e a parte correspondente ao endereço é carregada no MAR. Essa instrução pode ser obtida do IBR ou da memória, carregando a palavra correspondente no MBR e, a partir daí, no IBR, no IR e no MAR.

Por que usar essas vias indiretas? Essas operações são controladas por circuitos eletrônicos e resultam em transferências de dados. Para simplificar os circuitos eletrônicos, apenas um único registrador é empregado para especificar o endereço de memória para leitura ou para escrita e apenas um único registrador é utilizado como fonte ou destino do valor lido ou escrito.

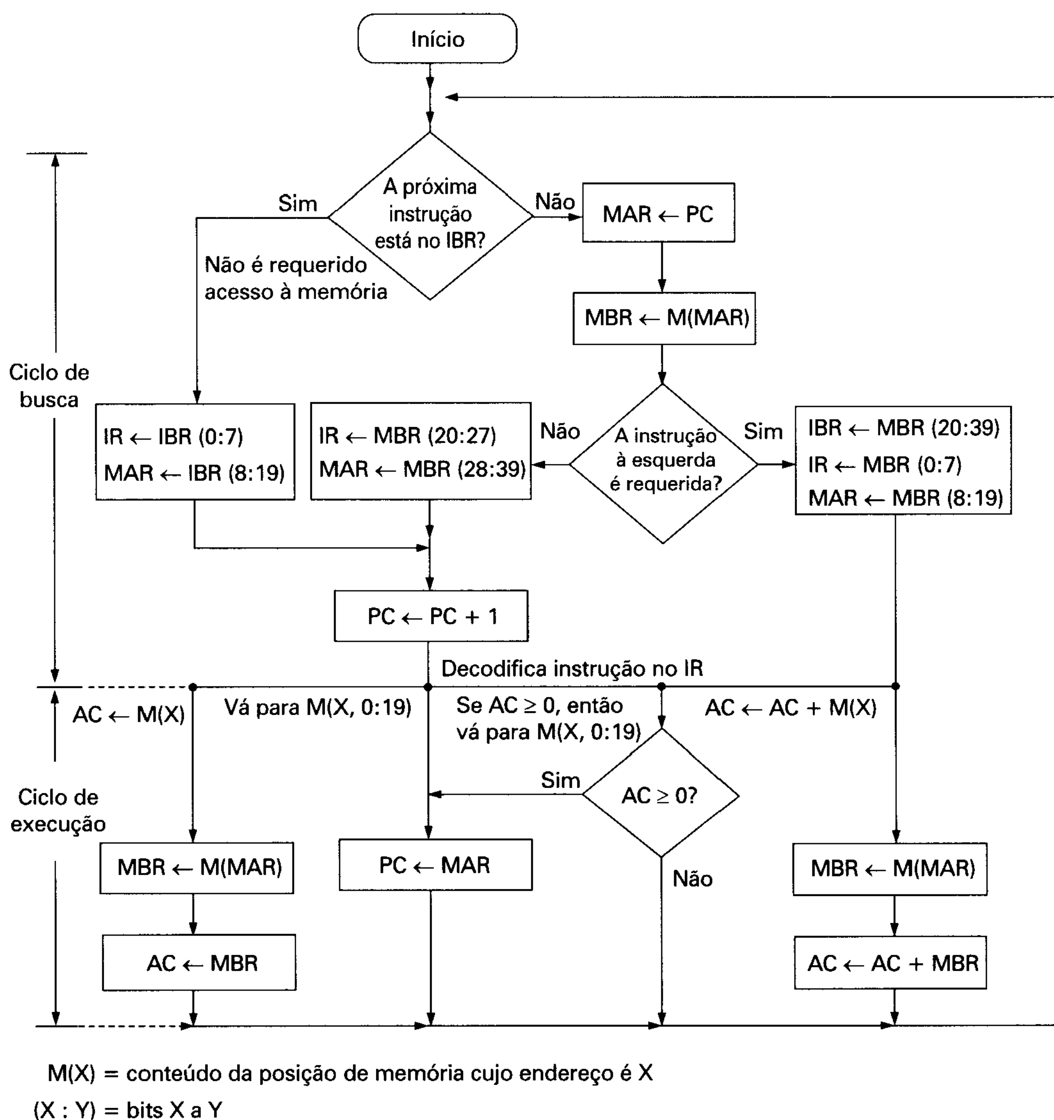


Figura 2.4 Fluxograma parcial da operação do IAS.

Depois que o código de uma operação é colocado no IR, inicia-se o *ciclo de execução*. O circuito de controle interpreta o código de operação e executa a instrução, enviando os sinais de controle apropriados, para fazer com que os dados sejam transferidos ou para que uma operação seja executada pela ULA.

O computador IAS tinha um total de 21 instruções, relacionadas na Tabela 2.1. Elas podem ser agrupadas como a seguir:

- **Transferência de dados:** os dados são transferidos entre a memória e os registradores da ULA ou entre dois registradores da ULA.
- **Desvio incondicional:** normalmente, a unidade de controle executa as instruções na seqüência em que se encontram na memória. Essa seqüência pode ser alterada por uma instrução de desvio. Isto é usado para executar seqüências de instruções repetidamente.
- **Desvio condicional:** o desvio é efetuado dependendo do teste de uma condição, o que permite a introdução de pontos de decisão.
- **Aritmética:** operações executadas pela ULA.
- **Alteração de endereço:** possibilita calcular endereços, utilizando a ULA, para então inseri-los em instruções armazenadas na memória. Isto permite ao programa uma considerável flexibilidade de endereçamento.

A Tabela 2.1 apresenta as instruções do IAS de forma simbólica, fácil de ler. Na verdade, cada instrução deve obedecer ao formato da Figura 2.2b. O código de operação (os primeiros 8 bits) especifica qual das 21 instruções deve ser executada. O endereço (os 12 bits restantes) determina qual das mil posições de memória é utilizada na execução da instrução.

A Figura 2.4 mostra diversos exemplos de execução de instrução pela unidade de controle. Note que cada operação requer diversos passos. Alguns desses passos são bastante elaborados. A operação de multiplicação exige 39 suboperações, uma para cada posição de bit, exceto para o bit de sinal!

Tabela 2.1 O conjunto de instruções do IAS

Tipo de instrução	Código de operação	Representação simbólica	Descrição
Transferência de dados	00001010	LOAD MQ	Transfere o conteúdo do registrador MQ para o acumulador AC
	00001001	LOAD MQ,M(X)	Transfere o conteúdo da posição de memória X para MQ
	00100001	STOR M(X)	Transfere o conteúdo do acumulador para a posição de memória X
	00000001	LOAD M(X)	Transfere M(X) para o acumulador
	00000010	LOAD – M(X)	Transfere – M(X) para o acumulador
	00000011	LOAD |M(X)|	Transfere o valor absoluto de M(X) para o acumulador
	00000100	LOAD – |M(X)|	Transfere – |M(X)| para o acumulador

(*continua*)

Tabela 2.1 O conjunto de instruções do IAS (*continuação*)

Tipo de instrução	Código de operação	Representação simbólica	Descrição
Desvio incondicional	00001101	JUMP M(X,0:19)	A próxima instrução a ser executada é buscada na metade esquerda de M(X)
	00001110	JUMP M(X,20:39)	A próxima instrução a ser executada é buscada na metade direita de M(X)
Desvio condicional	00001111	JUMP+(X,0:19)	Se o número no acumulador é um valor não-negativo, a próxima instrução a ser executada é buscada na metade esquerda de M(X)
	00010000	JUMP+M(X,20:39)	Se o número no acumulador é um valor não-negativo, a próxima instrução a ser executada é buscada na metade direita de M(X)
Aritmética	00000101	ADD M(X)	Soma M(X) a AC; armazena o resultado em AC
	00000111	ADD |M(X)|	Soma |M(X)| a AC; armazena o resultado em AC
	00000110	SUB M(X)	Subtrai M(X) de AC; armazena o resultado em AC
	00001000	SUB |M(X)|	Subtrai |M(X)| de AC; armazena o resto em AC
	00001011	MUL M(X)	Multiplica M(X) por MQ; armazena os bits mais significativos do resultado em AC, armazena os bits menos significativos em MQ.
	00001100	DIV M(X)	Divide AC por M(X); armazena o quociente em MQ e o resto em AC.
	00010100	LSH	Multiplica o acumulador por 2 (isto é, desloca os bits uma posição para a esquerda).
	00010101	RSH	Divide o acumulador por 2 (isto é, desloca os bits uma posição para a direita).
Alteração de endereço	00010010	STOR M(X,8:19)	Substitui o campo de endereço à esquerda de M(X) pelos 12 bits mais à direita de AC.
	00010011	STOR M(X,28:39)	Substitui o campo de endereço à direita de M(X) pelos 12 bits mais à direita de AC.

Computadores comerciais

Os anos 50 viram o nascimento da indústria de computadores com duas companhias, a Sperry e a IBM, dominando o mercado.

Em 1947, Eckert e Mauchly fundaram a Eckert-Mauchly Computer Corporation para fabricar computadores comercialmente. Sua primeira máquina de sucesso foi o UNIVAC I (*Universal Automatic Computer* — Computador Automático Universal), que foi financiado pelo Centro de Recenseamento para o censo de 1950. A Eckert-Mauchly Computer Corporation tornou-se parte da divisão UNIVAC da Sperry-Rand Corporation, que continuou a construir uma série de máquinas sucessoras.

O UNIVAC I foi o primeiro computador comercial de sucesso. Como o nome indica, ele tinha o propósito de servir tanto para aplicações científicas quanto para aplicações comerciais. O primeiro artigo que descreve esse sistema relatava, como amostra das tarefas que ele era capaz de executar, computações algébricas sobre matrizes, resolução de problemas estatísticos, cálculo de prêmios de seguro para uma companhia seguradora e solução de problemas logísticos.

O UNIVAC II, que possuía maior capacidade de memória e maior desempenho que o UNIVAC I, foi lançado no final dos anos 50 e ilustra tendências que permaneceram na indústria de computadores. A primeira é que os avanços da tecnologia permitiram que as companhias continuassem a desenvolver computadores cada vez mais poderosos e maiores. A segunda é que cada companhia procurava construir suas novas máquinas de modo que fossem *compatíveis* com as máquinas anteriores. Isso significa que os programas escritos para as máquinas mais antigas podiam ser executados nas máquinas mais novas. Essa estratégia é adotada na expectativa de manter os clientes; isto é, quando um cliente decidisse comprar uma nova máquina, provavelmente optaria por comprá-la do mesmo fabricante do seu antigo computador, para não perder o investimento já feito no desenvolvimento de programas.

A divisão UNIVAC iniciou também o desenvolvimento da série de computadores 1100, que seria sua linha de computadores de uso mais comum. O desenvolvimento dessa série mostra a distinção, existente anteriormente, entre computadores. O primeiro modelo, o UNIVAC 1103, e seus sucessores foram voltados para aplicações científicas que envolviam cálculos longos e complexos. Outras companhias dispunham de computadores mais voltados para aplicações comerciais que envolviam o processamento de grandes quantidades de textos. Essa distinção tornou-se menos evidente hoje em dia, mas foi bastante clara por muitos anos.

A IBM, que ajudou a construir o Mark I e era então o maior fabricante de dispositivos de processamento de cartões perfurados, lançou seu primeiro computador eletrônico programável, o 701, em 1953. O 701 foi, inicialmente, voltado para aplicações científicas (Bashe e outros, 1981). Em 1955, a IBM introduziu o modelo 702, que possuía várias características de hardware que o tornavam adequado para aplicações comerciais. Esses foram os primeiros de uma longa série de computadores 700/7000, que estabeleceram a IBM como o maior fabricante de computadores do mercado.

A segunda geração: transistores

A primeira grande mudança nos computadores eletrônicos veio com a substituição da válvula pelo transistor. O transistor é menor, mais barato e dissipa menos calor do que a válvula e, assim como uma válvula, também pode ser utilizado para a construção de computa-

dores. Ao contrário da válvula, que requer o uso de fios, placas de metal, cápsula de vidro e vácuo, o transistor é um *dispositivo de estado sólido,* feito de silício.

O transistor foi inventado na Bell Laboratories, em 1947, e iniciou uma revolução na indústria eletrônica nos anos 50. Entretanto, apenas no final da década de 50, computadores totalmente transistorizados tornaram-se comercialmente disponíveis. Mais uma vez, a IBM não foi a primeira companhia a lançar essa nova tecnologia. A NCR e, com maior sucesso, a RCA foram as pioneiras com o lançamento de pequenas máquinas transistorizadas. A IBM as seguiu de perto, com a série 7000.

O uso de transistores criou a *segunda geração* de computadores. É comum classificar os computadores em gerações, de acordo com a tecnologia básica de hardware empregada (Tabela 2.2). Cada nova geração é caracterizada por computadores com maior velocidade, maior capacidade de memória e menor tamanho que os computadores da geração anterior.

Tabela 2.2 Gerações de computadores

Geração	Datas aproximadas	Tecnologia	Velocidade típica (operações por segundo)
1	1946-1957	Válvula	40.000
2	1958-1964	Transistor	200.000
3	1965-1971	Integração em baixa e média escalas	1.000.000
4	1972-1977	Integração em grande escala	10.000.000
5	1978-	Integração em escala muito grande	100.000.000

Ocorreram também outras mudanças. Nos computadores da segunda geração, tanto a unidade lógica e aritmética quanto a unidade de controle eram mais complexas e os computadores já utilizavam linguagens de programação de alto nível e incluíam *software de sistema.*

Também merece destaque, na segunda geração, o surgimento da Digital Equipment Corporation (DEC). A DEC foi fundada em 1957 e lançou, nesse mesmo ano, seu primeiro computador, o PDP-1. Esse computador, juntamente com seu fabricante, deu início ao fenômeno do minicomputador, que se tornaria tão importante na terceira geração.

O IBM 7094

A partir da introdução da série 700, em 1952, até o lançamento do último modelo da série 7000, em 1964, essa linha de produtos da IBM passou por uma evolução típica dos produtos de computação. Os sucessivos membros da linha possuíam maior desempenho e capacidade e/ou custo mais baixo.

A Tabela 2.3 mostra essa tendência. O tamanho da memória principal, em múltiplos de 2^{10} palavras de 36 bits, cresceu de 2K ($1K = 2^{10}$) para 32K palavras, enquanto o tempo de acesso a uma palavra da memória, o *tempo de ciclo de memória,* caiu de 30 μs para 1,4 μs. O número de códigos de operação aumentou de 24 para 185.

A última coluna da Tabela 2.3 indica a velocidade relativa de execução da unidade central de processamento (CPU). O aumento da velocidade de processamento é devido a avanços da eletrônica (por exemplo, implementações que utilizam transistores são mais rápidas do que as que utilizam válvulas) e ao uso de circuitos mais complexos. Por exemplo, o IBM 7094

inclui um registrador secundário de instruções, usado para armazenar temporariamente a próxima instrução a ser executada. A unidade de controle busca, simultaneamente, duas palavras adjacentes na memória. Exceto no caso de uma instrução de desvio, o que não ocorre muito freqüentemente, isso faz com que a unidade de controle acesse a memória apenas na metade dos ciclos de instrução. Essa busca antecipada reduz significativamente o tempo médio do ciclo de instruções.

As informações contidas nas demais colunas da Tabela 2.3 serão explicadas no decorrer do texto.

A Figura 2.5 mostra uma configuração de grande porte, com muitos periféricos, para um IBM 7094, que constitui um computador bastante representativo da segunda geração (Bell, 1971a). Várias diferenças entre ele e o computador IAS podem ser destacadas. A mais importante é o uso de *canais de dados.* Um canal de dados é um módulo de E/S independente, com seu próprio processador e seu próprio conjunto de instruções. Em um sistema de computação com esses dispositivos, a CPU não executa instruções de E/S. Essas instruções são armazenadas na memória principal, sendo executadas por um processador especial no próprio canal de dados. A CPU inicia uma transferência de E/S enviando um sinal de controle ao canal de dados, que o instrui a efetuar uma seqüência de instruções armazenadas na memória. O canal de dados realiza sua tarefa independentemente da CPU, sinalizando-a quando a operação terminar. Esse arranjo evita um consumo considerável de CPU.

Outra característica nova é o *multiplexador,* que constitui o ponto central de conexão entre os canais de dados, a CPU e a memória. O multiplexador seleciona qual dispositivo, entre a CPU e os canais de dados, pode fazer acesso à memória. Isso permite que esses dispositivos executem de maneira independente.

A terceira geração: circuitos integrados

Um único transistor autônomo é denominado um *componente discreto.* Durante a década de 50 e o início dos anos 60, os equipamentos eletrônicos eram compostos basicamente de componentes discretos — transistores, resistores, capacitores e assim por diante. Esses componentes eram fabricados separadamente, encapsulados em seus próprios recipientes e soldados ou ligados com fios, por meio de uma técnica conhecida como *wire-up,* a placas de circuito, que eram então instaladas nos computadores, osciloscópios e outros equipamentos eletrônicos. Quando um dispositivo eletrônico requeria um transistor, um pequeno tubo de metal com uma peça de silício do tamanho de uma cabeça de alfinete tinha de ser soldado a uma placa de circuito. O processo completo de fabricação, desde o transistor até a placa de circuito, era caro e incômodo.

Isso começava a criar problemas na indústria de computadores. Os computadores do início da segunda geração continham cerca de 10 mil transistores. Esse número cresceu até centenas de milhares, tornando cada vez mais difícil a fabricação de máquinas novas, mais poderosas.

Em 1958, foi desenvolvida uma nova técnica que revolucionou os equipamentos eletrônicos e iniciou a era da microeletrônica: a invenção do circuito integrado. Esse circuito caracteriza a terceira geração de computadores. Nesta seção, fazemos uma breve introdução à tecnologia de circuitos integrados e, em seguida, examinamos os dois membros possivelmente mais importantes da terceira geração, ambos introduzidos no início dessa era: o sistema 360 da IBM e o PDP-8 da DEC.

Tabela 2.3 Exemplo de membros das séries 700/7000 da IBM.

Número do modelo	Primeira entrega	Tecnologia da CPU	Tecnologia da memória	Tempo de ciclo (μs)	Tamanho da memória (K)	Número de códigos de operação	Número de registradores indexadores	Hardware de ponto flutuante	Sobreposição de E/S (canais)	Sobreposição de busca de instruções	Velocidade (relativa ao 701)
701	1952	Válvula	Válvulas eletrostáticas	30	2-4	24	0	não	não	não	1
704	1955	Válvula	Núcleo de material ferro-magnético	12	4-32	80	3	sim	não	não	2,5
709	1958	Válvula	Núcleo de material ferro-magnético	12	32	140	3	sim	sim	não	4
7090	1960	Transistor	Núcleo de material ferro-magnético	2,18	32	169	3	sim	sim	não	25
7094 I	1962	Transistor	Núcleo de material -ferro magnéticO	2	32	185	7	sim (precisão dupla)	sim	sim	20
7094 II	1964	Transistor	Núcleo de material ferro-magnético	1,4	32	185	7	sim (precisão dupla)	sim	sim	50

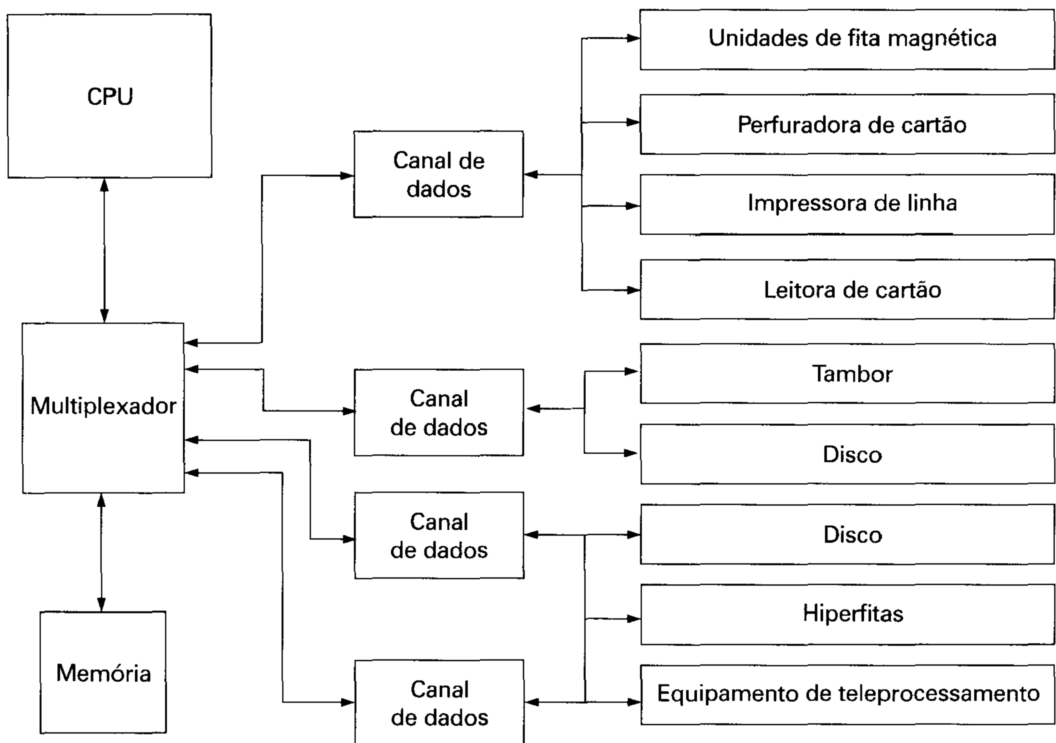


Figura 2.5 Uma configuração do IBM 7094.

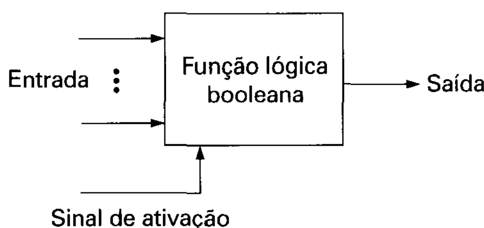
Microeletrônica

Microeletrônica significa, literalmente, 'eletrônica pequena'. Desde o início da eletrônica digital e da indústria de computadores, houve uma tendência persistente e consistente de redução do tamanho dos circuitos eletrônicos digitais. Antes de examinarmos as implicações e os benefícios dessa tendência, devemos fornecer algumas informações sobre a natureza da eletrônica digital. Uma discussão mais detalhada é apresentada no Apêndice A.

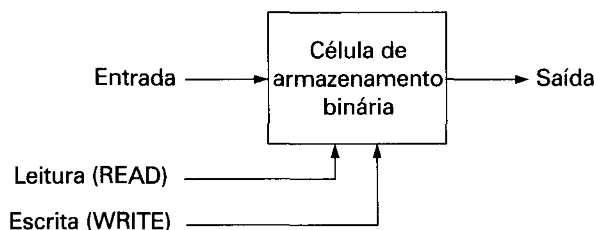
Como sabemos, os componentes básicos de um computador digital devem desempenhar funções de armazenamento, transferência, processamento e controle. Apenas dois tipos de componentes são necessários (Figura 2.6): portas lógicas e células de memória. Uma porta lógica é um dispositivo que implementa uma função lógica ou booleana, tal como: SE *A* E *B* SÃO VERDADEIROS, ENTÃO *C* É VERDADEIRO (porta lógica E). Esses dispositivos são chamados de portas lógicas porque controlam o fluxo de dados, assim como o fazem as portas de canais d'água. A célula de memória é um dispositivo que pode armazenar um valor binário, em um bit, isto é, o dispositivo pode estar, em cada instante, em um dos dois possíveis estados estáveis. Com a interconexão de um grande número desses dispositivos fundamentais podemos construir um computador. Esses dispositivos estão relacionados às quatro funções básicas descritas anteriormente do seguinte modo:

- **Armazenamento de dados:** fornecido pelas células de memória.
- **Processamento de dados:** fornecido pelas portas lógicas.

- **Transferência de dados:** os caminhos entre os componentes são usados para transferir dados da memória para a memória e da memória passando por portas lógicas para a memória.
- **Controle:** os caminhos entre os componentes podem também carregar sinais de controle. Por exemplo, uma porta lógica pode ter um ou dois dados de entrada, além de uma entrada de sinal de controle que ativa a porta. Quando o sinal de controle é ativado, a porta executa sua função sobre os dados de entrada e produz um dado de saída. Do mesmo modo, uma célula de memória armazena o bit do barramento de entrada, quando o sinal de controle WRITE é ativado, e coloca aquele bit no barramento de saída, quando o sinal de controle READ é ativado.



(a) Porta lógica



(b) Célula de memória

Figura 2.6 Elementos fundamentais de um computador.

Um computador consiste, portanto, em portas lógicas, células de memória e interconexões entre esses elementos. As portas lógicas e as células de memória são, por sua vez, construídas com componentes eletrônicos digitais simples.

O circuito integrado explora o fato de que componentes, como transistores, resistores e condutores, podem ser fabricados a partir de um único semicondutor como o silício. A fabricação de um circuito inteiro em uma pequena peça de silício, em vez de reunir em um mesmo circuito componentes discretos feitos de peças de silício separadas, é simplesmente uma extensão da tecnologia de estado sólido. É possível produzir simultaneamente milhares de transistores em uma única lâmina de silício. Além disso, esses transistores podem ser conectados entre si, por um processo de metalização, para formar os circuitos.

A Figura 2.7 representa os conceitos básicos de um circuito integrado. Uma fina *lâmina* de silício é dividida em uma matriz com pequenas áreas, cada uma medindo poucos milímetros quadrados. Um padrão de circuito idêntico é produzido em cada área, e a lâmina é dividida em pastilhas (*chips*). Cada pastilha é composta de muitas portas lógicas e/ou células de

memória, além de um grande número de pontos de conexão de entrada e saída. A pastilha é então empacotada em uma cápsula, que a protege e contém pinos para sua conexão com dispositivos externos. Várias pastilhas podem então ser interconectadas a uma placa de circuito impresso, produzindo circuitos maiores e mais complexos.

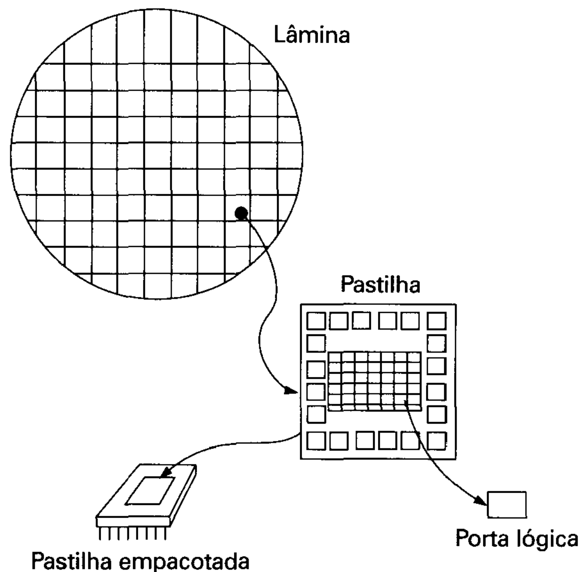


Figura 2.7 Relação entre lâmina, pastilha e porta lógica.

Inicialmente, era possível fabricar e empacotar em uma única pastilha apenas um pequeno número de portas lógicas ou células de memória. Esses primeiros circuitos integrados são chamados *circuitos com integração em baixa escala* (*small-scale integration* — SSI). Com o passar do tempo, foi possível empacotar mais e mais componentes em uma mesma pastilha. Esse aumento de densidade é ilustrado na Figura 2.8; esta é uma das mais notáveis tendências tecnológicas jamais registradas. A figura reflete a famosa *Lei de Moore*, proposta por Gordon Moore, um dos fundadores da Intel, em 1965 (Moore, 1965). Moore observou que o número de transistores que podiam ser impressos em uma única pastilha dobrava a cada ano e previu, corretamente, que esse crescimento permaneceria em um futuro próximo. Para a surpresa de muitos, inclusive de Moore, esse crescimento continuou, ano após ano e década após década. Nos anos 70, a taxa de crescimento diminuiu, com a duplicação ocorrendo a cada 18 meses, mas estabilizou-se desde então.

As conseqüências da Lei de Moore são profundas:

1. O custo de uma pastilha permaneceu praticamente inalterado ao longo desse período de rápido crescimento da sua densidade. Isso significa que o custo de implementação da lógica do computador e do seu circuito de memória caiu de maneira dramática.
2. Como as portas lógicas e as células de memória eram empacotadas cada vez mais próximas umas das outras nas pastilhas e em maior número, o caminho elétrico entre elas encurtava, aumentando a velocidade de operação.

3. O computador ficou cada vez menor, tornando-se mais conveniente para ser usado em diversos ambientes.
4. Ocorreu uma grande redução do consumo de energia elétrica e da necessidade de resfriamento do equipamento.
5. As interconexões em um circuito integrado são muito mais confiáveis do que as conexões soldadas. Com maior número de circuitos em cada pastilha, o número de conexões requeridas entre pastilhas é muito menor.

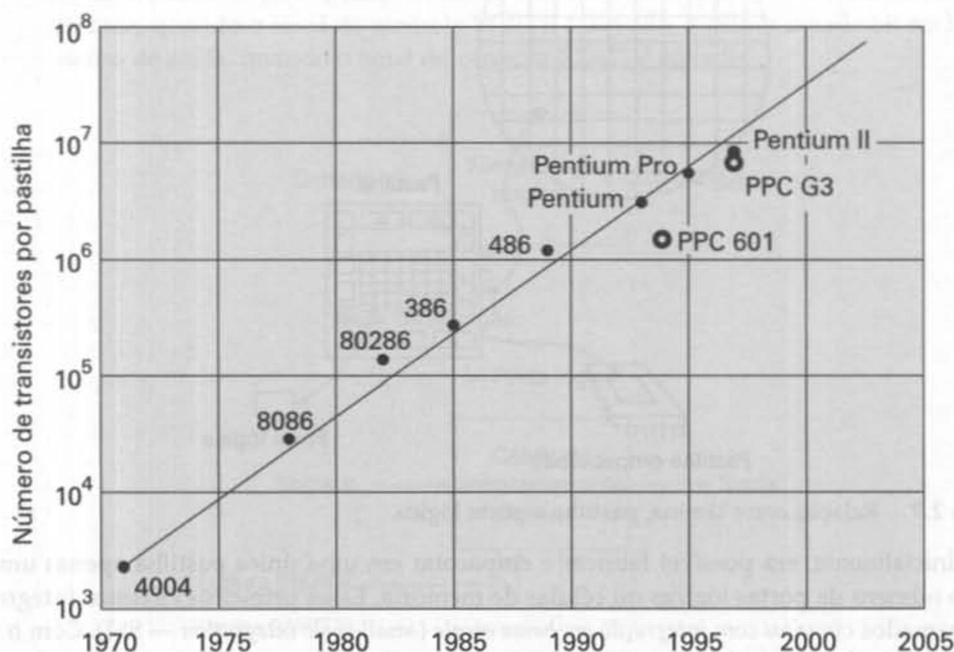


Figura 2.8 Crescimento do número de transistores da CPU.

Sistema 360 da IBM

Em 1964, a IBM possuía um forte controle do mercado de computadores, com sua série de máquinas 7000. Nesse ano, a IBM anunciou o Sistema 360 (S/360 ou Sistema/360), uma nova família de computadores. Embora o anúncio em si não fosse uma surpresa, ele continha algumas novidades desagradáveis para seus clientes: a linha de produtos 360 era incompatível com as máquinas IBM mais antigas. Assim, seria difícil a transição dos atuais clientes para o modelo 360. Esse era um passo ousado, que a IBM sentiu ser necessário para evitar algumas restrições da arquitetura 7000 e produzir um sistema capaz de evoluir com a nova tecnologia de circuitos integrados (Padegs, 1981; Gifford, 1987). A estratégia foi bem-sucedida tanto financeira quanto tecnicamente. O 360 foi o sucesso da década e solidificou a IBM como o maior fabricante e vendedor de computadores, com uma parcela de mais de 70% do mercado. A arquitetura do 360 permanece até hoje, com algumas modificações e extensões, como a arquitetura dos computadores de grande porte da IBM. Vários exemplos baseados nessa arquitetura são utilizados no decorrer deste livro.

O Sistema 360 foi a primeira família planejada de computadores da indústria. A família cobria uma ampla gama de desempenho e de custo. A Tabela 2.4 indica algumas das características principais dos vários modelos existentes em 1965 (cada membro da família é diferenciado pelo número do modelo). Os modelos eram compatíveis entre si, uma vez que um programa escrito para um modelo podia ser executado por um outro modelo da série, sendo diferente apenas o tempo requerido para sua execução.

O conceito de uma família de computadores compatíveis era novo e extremamente bem-sucedido. Um cliente com orçamento e exigências mais modestos poderia começar com o Modelo 30, relativamente barato. Mais tarde, caso suas necessidades de computação crescessem, ele poderia migrar para uma máquina mais rápida e com maior capacidade de memória, sem sacrificar o investimento já feito em desenvolvimento de software. As características de uma família de computadores são as seguintes:

- **Conjunto de instruções idêntico ou semelhante:** em muitos casos, o mesmo conjunto de instruções de máquina é usado em todos os membros da família. Portanto, um programa que é executado em uma máquina pode também ser executado em qualquer outra. Em alguns casos, os modelos mais simples da família possuem um conjunto de instruções que constitui um subconjunto das instruções usadas nos computadores de topo de linha da família. Isso significa que os programas podem migrar para computadores de nível mais alto, mas não para níveis mais baixos.
- **Sistema operacional idêntico ou semelhante:** o mesmo sistema operacional básico está disponível em todos os membros da família. Em alguns casos, novos recursos são incorporados aos computadores de níveis superiores.
- **Velocidade crescente:** a taxa de execução de instruções aumenta dos membros de níveis mais baixos para os de níveis mais altos da família.
- **Número crescente de portas de E/S:** aumento do número de portas de entrada e saída, dos membros de níveis mais baixos para os de níveis mais altos da família.
- **Capacidade de memória crescente:** maior capacidade de memória nos computadores de níveis mais altos da família.
- **Custo crescente:** custo mais alto dos computadores, na direção dos membros de níveis mais baixos para os de níveis mais altos da família.

Tabela 2.4 Características principais da família Sistema 360

Característica	Modelo 30	Modelo 40	Modelo 50	Modelo 65	Modelo 75
Capacidade máxima de memória (bytes)	64K	256K	256K	512K	512K
Taxa de transferência de dados da memória (Mbytes/s)	0,5	0,8	2,0	8,0	16,0
Tempo de ciclo do processador (μs)	1,0	0,625	0,5	0,25	0,2
Velocidade relativa	1	3,5	10	21	50
Número máximo de canais de dados	3	3	4	6	6
Taxa máxima de transferência por canal (Kbytes/s)	250	400	800	1.250	1.250

Como o conceito de família de computadores podia ser implementado? O aumento de desempenho era obtido, principalmente, em razão de três fatores: a velocidade básica, o tamanho e o grau de simultaneidade (Stevens, 1964). Por exemplo, uma maior velocidade de execução de uma dada instrução podia ser obtida pelo uso de um conjunto de circuitos mais complexos na ULA, que possibilitasse a execução de suboperações em paralelo. Outro mecanismo para aumentar a velocidade era a ampliação da quantidade de dados (largura do fluxo de dados) transferidos simultaneamente entre a memória principal e a CPU. No modelo 30, apenas 1 byte (8 bits) podia ser lido da memória de cada vez; já no Modelo 70, 8 bytes podiam ser transferidos ao mesmo tempo.

O Sistema 360 influenciou não apenas o futuro da IBM, mas também toda a indústria de computadores. Muitas de suas características se tornaram padrão em outros computadores de grande porte.

DEC PDP-8

No mesmo ano em que a IBM lançou o primeiro modelo da família 360, ocorreu outro lançamento importante: o do PDP-8 da DEC. Em uma época em que a maioria dos computadores precisava de uma sala com ar-condicionado, o PDP-8 (denominado pela indústria um minicomputador) era suficientemente pequeno para poder ser colocado em cima de uma bancada de laboratório ou incorporado a outro equipamento. Embora não tivesse um desempenho comparável aos dos computadores de grande porte, seu custo, de 16 mil dólares, era suficientemente baixo para permitir que cada técnico de laboratório tivesse um. Por outro lado, os computadores da série Sistema 360, lançados apenas alguns meses antes, custavam centenas de milhares de dólares.

O baixo custo e o tamanho reduzido do PDP-8 possibilitavam que outros fabricantes o utilizassem como parte integrante de seus próprios sistemas. Esses fabricantes passaram a ser conhecidos como fabricantes originais de equipamentos (*original equipment manufacturers* — OEMs), e o mercado OEM tornou-se, e permanece até hoje, um dos maiores segmentos do mercado de computadores.

O PDP-8 tornou-se imediatamente um sucesso e deu muitos lucros à DEC. Essa máquina, assim como outros membros da família PDP-8 que a seguiram (veja a Tabela 2.5), atingiu níveis de produção só comparáveis, na época, aos computadores IBM, com cerca de 50 mil máquinas vendidas nos 12 anos seguintes. Segundo a própria história oficial da DEC, o PDP-8 "estabeleceu o conceito de minicomputadores, abrindo caminho para uma indústria multibilionária". A DEC passou a ser o maior vendedor de minicomputadores. Na época em que o PDP-8 chegou ao final de sua vida útil, a DEC era o segundo maior fabricante de computadores, atrás apenas da IBM.

Ao contrário da arquitetura de comutação centralizada (Figura 2.5) utilizada pela IBM em seus sistemas 700/7000 e 360, os últimos modelos do PDP-8 usavam uma estrutura que é praticamente universal nos minicomputadores e microcomputadores de hoje: a estrutura de barramento. Essa estrutura é ilustrada na Figura 2.9. O barramento do PDP-8, chamado Omnibus, consistia em um caminho de 96 sinais distintos, empregados para carregar sinais de controle, de endereços e de dados. Como todos os componentes do sistema compartilham um mesmo conjunto de sinais, a utilização do barramento deve ser controlada pela CPU.

Tabela 2.5 Evolução do PDP-8 (Voelker, 1988)

Modelo	Primeira venda	Custo do processador + memória com 4K palavras de 12 bits (em milhares de dólares)	Transferência de dados da memória (palavras/µs)	Volume (pés cúbicos)	Inovações e melhorias
PDP-8	4/65	16,2	1,26	8,0	*Wire-wrapping* automático
PDP-8/5	9/66	8,79	0,08	3,2	Implementação de instrução serial
PDP-8/1	4/68	11,6	1,34	8,0	Circuito integrado em média escala (MSI)
PDP-8/L	11/68	7,0	1,26	2,0	Gabinete menor
PDP-8/E	3/71	4,99	1,52	2,2	Barramento Omnibus
PDP-8/M	6/72	3,69	1,52	1,8	Gabinete com metade do tamanho
PDP-8/A	1/75	2,6	1,34	1,2	Memória de semicondutor; processador de ponto flutuante

Essa arquitetura é altamente flexível, permitindo que novos módulos sejam acoplados ao barramento, de modo que se obtenha diferentes configurações. A Figura 2.10 mostra uma configuração de grande porte do PDP-8/E.

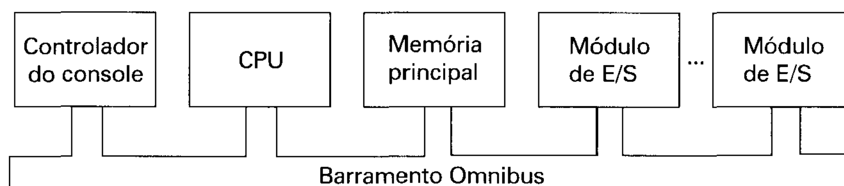


Figura 2.9 Estrutura do barramento do PDP-8.

Últimas gerações

A partir da terceira geração de computadores, existe um menor consenso sobre a definição das demais gerações de computadores. A Tabela 2.2 sugere a existência de uma quarta e uma quinta gerações, com base na evolução da tecnologia de circuitos integrados. Com a introdução de integração em grande escala (*large-scale integration* — LSI), mais de mil componentes podem ser colocados em uma única pastilha de circuito integrado. A integração em escala muito grande (*very-large-scale integration* — VLSI) atingiu mais de 10 mil componentes por pastilha, e as pastilhas VLSI atuais podem conter mais de 100 mil componentes.

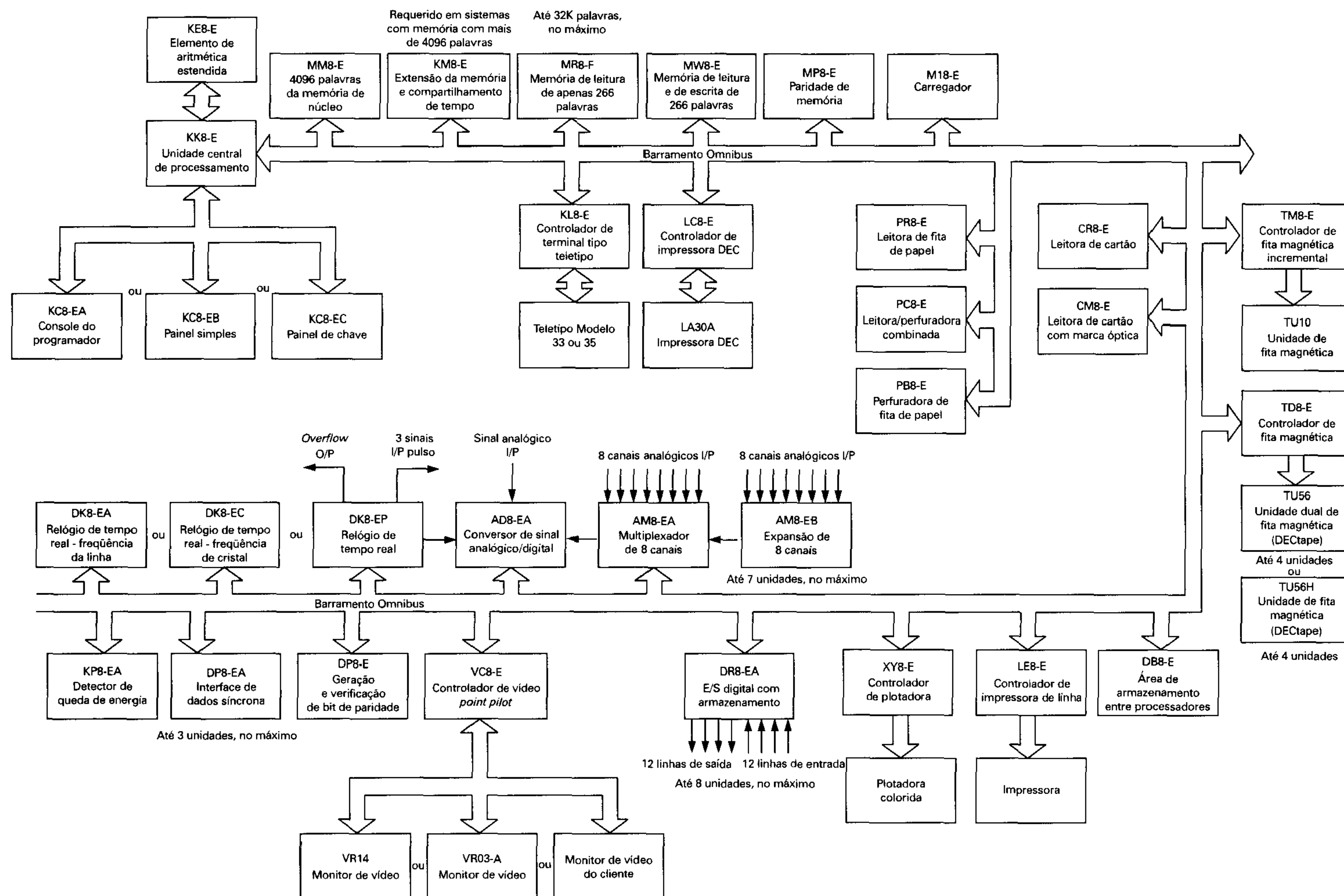


Figura 2.10 Diagrama de blocos do sistema PDP-8/E.

Com o rápido avanço da tecnologia, a introdução significativa de novos produtos e a importância do software e das comunicações, tanto quanto do hardware, a classificação em gerações torna-se menos clara e menos significativa. Pode-se dizer que a aplicação comercial dos novos desenvolvimentos representou uma grande mudança no início dos anos 70 e que os resultados dessas mudanças ainda estão sendo sentidos. Nesta seção, mencionamos dois desses resultados mais importantes.

Memória de semicondutores

A primeira aplicação da tecnologia de circuito integrado em computadores foi na construção do processador (com a implementação da unidade de controle e da unidade lógica e aritmética) a partir de pastilhas de circuito integrado. Descobriu-se também que essa mesma tecnologia poderia ser usada para construir memórias.

Nos anos 50 e 60, a maioria das memórias de computadores era construída a partir de pequenos anéis de material ferro-magnético, cada um com cerca de 1,5 mm de diâmetro. Esses anéis eram organizados em grades com fios finos, suspensos em pequenas telas dentro do computador. Quando magnetizado em um certo sentido, um anel (também chamado de *núcleo*) representa o dígito 1 (*um*); quando magnetizado no outro sentido, ele representa o dígito 0 (*zero*). A memória de núcleo magnético era bastante rápida; um bit armazenado na memória era lido em apenas um milionésimo de segundo. Mas essa memória era cara, volumosa e a leitura era uma operação destrutiva: o simples ato de ler um núcleo apagava os dados nele armazenados. Era necessário, assim, utilizar circuitos que armazenassem novamente os dados, cada vez que fossem lidos.

Em 1970, Fairchild produziu a primeira memória de semicondutores, relativamente volumosa se comparada às memórias existentes hoje em dia. A pastilha de memória tinha quase o tamanho de um único núcleo e podia conter 256 bits de memória. A leitura não era destrutiva e era muito mais rápida do que os núcleos. Apenas 70 bilionésimos de segundo eram necessários para ler um bit. Entretanto, o custo por bit era mais alto do que no núcleo magnético.

Em 1974, o preço por bit da memória de semicondutores tornou-se menor do que o preço por bit da memória de núcleo magnético. Depois disso, houve um contínuo e rápido declínio do custo de memória, acompanhado por um correspondente aumento de densidade da memória física. Isso abriu caminho para a construção, alguns anos depois, de máquinas menores, mais rápidas e com capacidade de memória semelhante às das máquinas mais poderosas e mais caras. O desenvolvimento da tecnologia de memória, juntamente com o desenvolvimento da tecnologia de processadores, discutida a seguir, mudou a natureza dos computadores, em menos de uma década. Embora ainda existissem computadores volumosos e caros, os computadores foram apresentados para o "usuário final", com as máquinas de escritórios e os computadores pessoais.

Desde 1970, a memória de semicondutores passou por dez gerações: 1K, 4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M e, enquanto era escrito este livro, 256M bits em uma única pastilha ($1K = 2^{10}$, $1M = 2^{20}$). Em cada geração, a densidade da memória aumentou quatro vezes, acompanhada por um declínio do custo por bit e um declínio do tempo de acesso.

Microprocessadores

Ao mesmo tempo em que a densidade de elementos nas pastilhas de memória continuava a crescer, também crescia continuamente a densidade de elementos nas pastilhas do processador. Com o passar do tempo, mais e mais elementos foram colocados em uma pastilha e, portanto, menor número de pastilhas era necessário para construir um único processador.

Um marco importante foi o desenvolvimento do Intel 4004, em 1971. O 4004 foi a primeira pastilha a conter *todos* os componentes de uma CPU: havia nascido o microprocessador.

O 4004 era capaz de somar dois números de 4 bits e efetuava multiplicação por meio de somas repetidas. Embora fosse muito primitivo em relação aos padrões atuais, ele marcou o começo de uma evolução contínua na capacidade e no poder de processamento dos microprocessadores.

Essa evolução é percebida mais facilmente considerando-se o número de bits que um processador é capaz de trabalhar de cada vez. Embora não exista uma medida exata para esse parâmetro, a largura do barramento de dados talvez seja a mais significativa: o número de bits de dados que pode ser recebido ou enviado pelo processador de cada vez. Outra medida possível é o número de bits do registrador acumulador ou do conjunto de registradores de propósito geral. Essas medidas freqüentemente coincidem umas com as outras, mas nem sempre. Por exemplo, foram fabricados alguns microprocessadores que operavam com valores de 16 bits em seus registradores, mas podiam ler e escrever apenas 8 bits de cada vez.

O próximo passo importante na evolução dos microprocessadores ocorreu com a introdução do Intel 8008, em 1972. Esse foi o primeiro microprocessador de 8 bits e era quase duas vezes mais complexo do que o 4004.

Nenhum desses acontecimentos teria o impacto do evento seguinte: a introdução, em 1974, do Intel 8080. Este foi o primeiro microprocessador de propósito geral. Enquanto o 4004 e o 8008 foram projetados para aplicações específicas, o 8080 foi projetado para ser a CPU de um microcomputador de propósito geral. Como o 8008, o 8080 era também um microprocessador de 8 bits. Entretanto, ele era mais rápido, possuía um conjunto de instruções mais rico e uma grande capacidade de endereçamento à memória.

Nessa mesma época, começaram a ser desenvolvidos os microprocessadores de 16 bits. No entanto, apenas no final dos anos 70 surgiram microprocessadores de 16 bits de propósito geral. Um deles foi o 8086. Seguindo essa tendência, a Bell Laboratories e a Hewlett-Packard desenvolveram, em 1981, microprocessadores de 32 bits de uma única pastilha. A Intel lançou seu microprocessador de 32 bits, o 80386, em 1985 (veja a Tabela 2.6).

Tabela 2.6 Evolução dos microprocessadores da Intel

(a) Processadores da década de 70

	Microprocessador 4004	Microprocessador 8008	Microprocessador 8080	Microprocessador 8086	Microprocessador 8088
Data de lançamento	15/11/1971	1/4/1972	1/4/1974	8/6/1978	1/6/1979
Velocidade de relógio	108 KHz	108 KHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8 MHz
Largura do barramento	4 bits	8 bits	8 bits	16 bits	8 bits
Número de transistores (mícrons)	2.300 (10)	3.500	6.000 (6)	29.000 (3)	29.000 (3)
Memória endereçável	640 bytes	16 kbytes	64 kbytes	1 megabyte	1 megabyte
Memória virtual	—	—	—	—	—

(b) Processadores da década de 80

	Microprocessador 80286	Microprocessador Intel 386TM DX	Microprocessador Intel 386TM SX	Microprocessador Intel 486TM DX CPU
Data de lançamento	1/2/1982	17/10/1985	16/6/1988	10/4/1989
Velocidade de relógio	6 MHz – 12,5 MHz	16 MHz – 33 MHz	16 MHz – 33 MHz	25 MHz – 50 MHz
Largura do barramento	16 bits	32 bits	16 bits	32 bits
Número de transistores (mícrons)	134.000 (1,5)	275.000 (1)	275.000 (1)	1,2 milhão (0,8 – 1)
Memória endereçável	16 megabytes	4 gigabytes	4 gigabytes	4 gigabytes
Memória virtual	1 gigabyte	64 terabytes	64 terabytes	64 terabytes

(c) Processadores da década de 90

	Microprocessador Intel 486TMSX	Processador Pentium®	Processador Pentium® Pro	Processador Pentium® II
Data de lançamento	22/4/1991	22/3/1993	1/11/1995	7/5/1997
Velocidade de relógio	16 MHz – 33 MHz	60 MHz – 166 MHz	150 MHz – 200 MHz	200 MHz – 300 MHz
Largura do barramento	32 bits	32 bits	64 bits	64 bits
Número de transistores (mícrons)	1,185 milhão (1)	3,1 milhões (0,8)	5,5 milhões (0,6)	7,5 milhões
Memória endereçável	4 megabytes	4 gigabytes	64 gigabytes	64 gigabytes
Memória virtual	64 gigabytes	64 terabytes	64 terabytes	64 terabytes

Fonte: Intel Corp. `http://www.intel.com/intel/museum/25anniv/hof/tspecs.htm`

2.2 PROJETO QUE VISA AO DESEMPENHO

O custo dos sistemas de computação continuava a cair dramaticamente, ano após ano, enquanto o desempenho e a capacidade desses sistemas cresciam na mesma proporção. Hoje é possível adquirir um computador pessoal de menos de mil dólares com poder de processamento superior ao dos computadores de grande porte da IBM de dez anos atrás. Dentro desse computador pessoal, incluindo o microprocessador, a memória e outras pastilhas, existem aproximadamente 100 milhões de transistores. Não é possível comprar 100 milhões de unidades de qualquer outra coisa pelo mesmo preço, ou seja, nada custa tão pouco quanto um transistor.

O poder de processamento é, portanto, virtualmente 'gratuito'. Essa contínua revolução tecnológica possibilitou o desenvolvimento de aplicações com recursos e complexidade espantosos. Exemplos de aplicações comuns que demandam o alto poder de processamento dos microprocessadores atuais incluem:

- Processamento de imagem
- Reconhecimento de voz
- Videoconferência
- Aplicações multimídia
- Arquivos de anotações em voz e vídeo

As estações de trabalho permitem a execução de aplicações científicas e de engenharia altamente sofisticadas, assim como de sistemas de simulação, além de possibilitar a utilização de princípios de trabalho cooperativo a aplicações que manipulam imagem e vídeo. Além disso, aplicações comerciais dependem de estações servidoras de capacidade de processamento cada vez maior, para armazenar bancos de dados e processar transações e para gerenciar grandes redes do tipo cliente-servidor que substituíram os grandes centros de computação de grande porte do passado.

O mais fascinante nisso tudo, do ponto de vista de arquitetura e organização de computadores, é que, se por um lado os blocos básicos dos milagrosos computadores de hoje continuam sendo praticamente os mesmos do computador IAS de quase 50 anos atrás, por outro as técnicas para obter o máximo desempenho a partir dos recursos disponíveis tornaram-se cada vez mais sofisticadas.

Essas observações serviram como diretrizes para a estrutura dada a este livro. Ao longo de todo o livro, a descrição de cada um dos componentes de um computador procura atender a dois objetivos. Primeiro, entender a funcionalidade básica de cada área em questão e, segundo, explorar técnicas para obter o máximo desempenho. No restante desta seção, realçamos alguns dos principais fatores que justificam a necessidade de projetos que visem a um alto desempenho.

Velocidade do microprocessador

O impressionante poder de computação do Pentium ou do PowerPC revela a incansável busca dos fabricantes de processadores por alta velocidade. A evolução dessas máquinas continua a confirmar a Lei de Moore, citada anteriormente. Enquanto vigorar essa lei, os fabricantes poderão lançar uma nova geração de pastilhas a cada três anos — com número de transistores quatro vezes maior do que na geração anterior. No caso das pastilhas de memória, isso quadruplicou, a cada três anos, a capacidade da memória dinâmica de acesso aleató-

rio (*dynamic random-access memory* — DRAM), que ainda hoje é a tecnologia básica de implementação de memória principal de computadores. Desde que a Intel lançou sua família X86 em 1978, o desempenho dos microprocessadores aumentou quatro ou cinco vezes a cada três anos, em razão do aparecimento de novos circuitos e do aumento de velocidade, obtido pela redução das distâncias entre os componentes.

Mas a grande velocidade dos microprocessadores não será aproveitada, a menos que eles possam ser alimentados com um fluxo constante de instruções para execução. Qualquer perturbação no fluxo de instruções deteriora a velocidade de processamento. Por isso, enquanto os fabricantes de pastilhas se preocupam em fabricar pastilhas de densidade cada vez maior, os projetistas de processadores precisam obter técnicas cada vez mais elaboradas para alimentar o 'monstro' com instruções, ou seja, para manter o fluxo de execução de instruções. Entre as novas técnicas embutidas nos processadores atuais, estão as seguintes:

- **Previsão de desvios:** o processador examina instruções a serem executadas mais à frente e prevê quais desvios ou grupos de instruções têm maior probabilidade de ser processados. Se, na maioria das vezes, essa previsão for feita corretamente, a busca de instruções na memória pode ser antecipada, e essas instruções podem ser temporariamente armazenadas no processador, de modo que aumentem a taxa de execução de instruções pelo processador. Estratégias mais elaboradas são capazes de prever não apenas o próximo desvio, mas diversos desvios à frente. Portanto, a técnica de previsão de desvios pode aumentar sensivelmente o conjunto de instruções disponíveis para execução pelo processador.
- **Análise do fluxo de dados:** o processador verifica quais instruções dependem de resultados ou dados de outras instruções, determinando uma seqüência otimizada para a execução de instruções. As instruções são escalonadas de modo que sejam executadas tão logo estejam prontas, independentemente da ordem original em que ocorrem no programa. Essa técnica evita atrasos na execução de instruções.
- **Execução especulativa:** usando técnicas de previsão de desvios e de análise do fluxo de dados, alguns processadores executam instruções de maneira especulativa, antes que elas ocorram na seqüência de execução do programa, armazenando os resultados obtidos em registradores de armazenamento temporário. Se a maioria das instruções executadas antecipadamente realmente for necessária, será possível manter o processador ocupado o maior tempo possível.

Essas e outras técnicas elaboradas são necessárias para obter processadores poderosos, explorando ao máximo sua alta velocidade.

Balanceamento do desempenho

Enquanto a velocidade do processador cresceu assustadoramente, outros componentes críticos do computador não acompanharam essa evolução. Em razão disso, é preciso buscar um bom balanceamento de desempenho: um ajuste da organização e da arquitetura para compensar as diferenças de capacidade dos vários componentes.

O problema mais crítico decorrente dessas diferenças de desempenho ocorre na interface entre o processador e a memória principal. Considere a evolução das características de memórias e processadores apresentada na Figura 2.11. Enquanto a velocidade do processador e a capacidade da memória cresceram rapidamente, não houve correspondente evolução da

taxa de transferência de dados entre a memória principal e o processador. A interface entre processador e memória principal é a rota de informações mais importante do computador, pois é responsável pelo tráfego de um constante fluxo de dados e instruções entre as pastilhas da memória e do processador. Se a memória ou a interface não são capazes de atender à demanda do processador na velocidade adequada, o processador fica bloqueado, em estado de espera, desperdiçando valioso tempo de processamento.

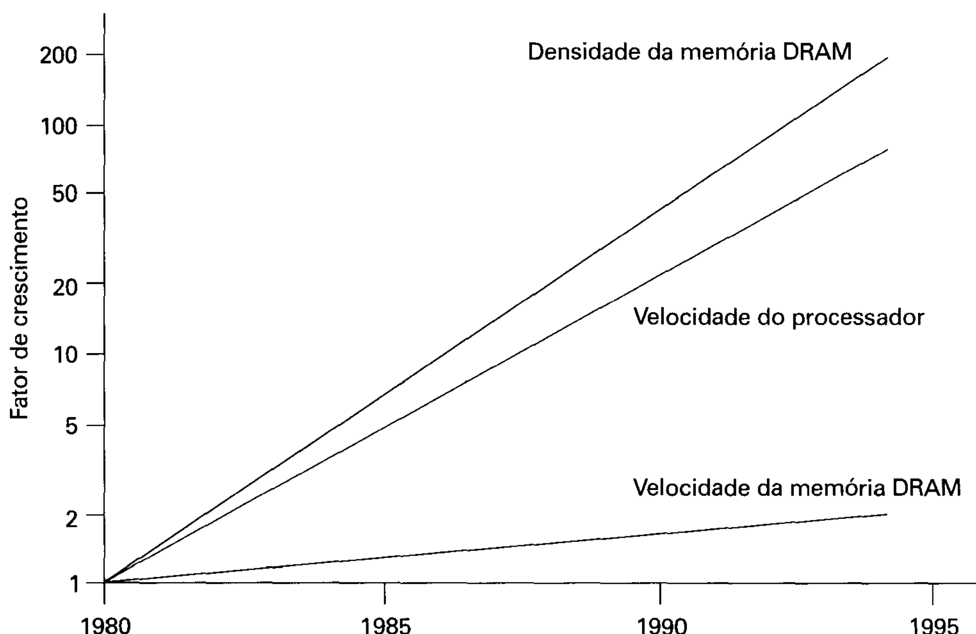


Figura 2.11 Evolução das características de memórias DRAM e processadores.

Os efeitos dessa tendência de evolução são mostrados claramente na Figura 2.12. A quantidade de memória principal requerida vem crescendo rapidamente, mas a densidade de memórias DRAM aumenta ainda mais rápido. O resultado é que, em média, o número de memórias DRAM por sistema vem caindo. As linhas em negrito na figura mostram que, para uma memória de tamanho fixo, o número necessário de memórias DRAM está em declínio. Entretanto, isso tem influência sobre a taxa de transferência de dados, uma vez que um menor número de memórias DRAM diminui as oportunidades para transferência de dados em paralelo. As partes sombreadas mostram que, para um tipo particular de sistema, o tamanho da memória principal vem aumentando vagarosamente, enquanto o número de memórias DRAM diminui.

Um projetista de sistema pode atacar esse problema de várias maneiras, e todas se baseiam em projetos de computadores atuais. Algumas delas são:

- Ampliar o número de bits obtidos em cada acesso à memória, aumentando-se a largura das memórias DRAM em vez de sua capacidade e utilizando barramentos de dados mais largos.

- Mudar a interface da memória DRAM para torná-la mais eficiente, usando uma memória cache ou outro esquema de armazenamento temporário na pastilha da memória DRAM.
- Reduzir a freqüência de acesso à memória, incorporando estruturas de memórias cache eficientes e complexas entre o processador e a memória principal. Isso inclui o uso de memórias cache tanto na pastilha do processador como fora dele.
- Aumentar a largura de banda da conexão entre processadores e memórias usando barramento de alta velocidade e uma hierarquia de barramentos para estruturar o fluxo de dados e armazenar os dados temporariamente.

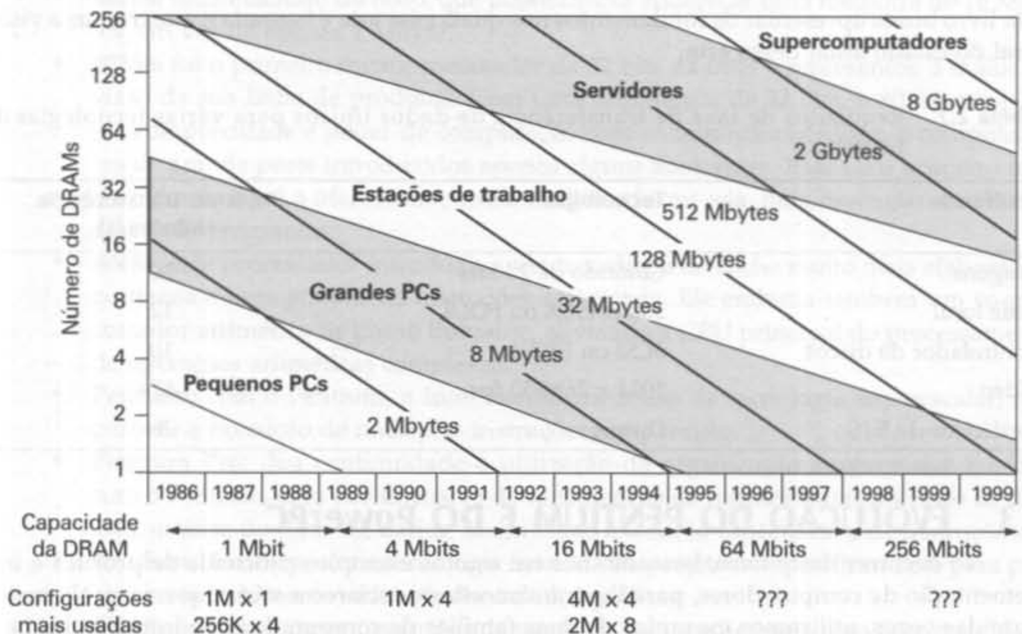


Figura 2.12 Tendências no uso de memórias DRAM (Przybylski, 1994).

Outra área de projeto importante é a dos dispositivos de E/S. À medida que os computadores se tornam mais rápidos e poderosos, foram desenvolvidas aplicações mais sofisticadas que fazem uso de periféricos com uma grande demanda por E/S. A Tabela 2.7 ilustra alguns exemplos de dispositivos periféricos típicos utilizados em computadores pessoais e estações de trabalho. Esses dispositivos criam enormes demandas de transferência de dados. Embora a atual geração de processadores seja capaz de processar os dados produzidos por esses dispositivos, temos ainda o problema da transferência desses dados entre os periféricos e o processador. As estratégias utilizadas para resolver esse problema incluem o uso de estruturas de cache e áreas de armazenamento temporário, assim como o uso de barramentos de alta velocidade e de estruturas de barramento mais elaboradas. Além disso,.o uso de configurações com múltiplos processadores também pode ajudar a atender às demandas de E/S.

A questão-chave do projeto é o balanceamento. Os projetistas esforçam-se constantemente para obter equilíbrio entre as demandas de transferência de dados e processamento dos

diversos componentes do processador, da memória principal, dos dispositivos de E/S e das estruturas de interconexão. O projeto deve ser constantemente avaliado, de modo que se adapte a dois fatores que evoluem constantemente:

- A taxa de variação de desempenho nas diversas áreas da tecnologia (processador, barramentos, memória, periféricos) difere em muito de um tipo de elemento para outro.
- O desenvolvimento de novas aplicações e novos dispositivos periféricos muda constantemente a natureza da demanda no sistema, em função do perfil de instruções típicas e do padrão de acesso a dados.

O projeto de um computador constitui, portanto, uma arte constantemente em evolução. Este livro busca apresentar os fundamentos nos quais essa arte é baseada e mostrar uma visão geral do estado atual dessa arte.

Tabela 2.7 Requisitos de taxa de transferência de dados típicos para várias tecnologias de periféricos

Periférico	Tecnologia	Taxa de transferência (Mbytes/s)
Imagens	Colorido de 24 bits	30
Rede local	100 BASEX ou FDDI	12
Controlador de discos	SCSI ou P1394	10
Vídeo	1024 × 768@30 fps	67+
Periféricos de E/S	Outros	5+

2.3 EVOLUÇÃO DO PENTIUM E DO PowerPC

No decorrer deste livro, baseamo-nos em muitos exemplos concretos de projetos e implementação de computadores, para ilustrar conceitos e esclarecer várias questões. Na maior parte das vezes, utilizamos exemplos de duas famílias de computadores: o Intel Pentium e o PowerPC. O Pentium representa o resultado de décadas de esforço de projeto em computadores com um conjunto complexo de instruções (*complex instruction set computers* — CISC). Ele incorpora princípios de projetos sofisticados, encontrados apenas em computadores de grande porte e supercomputadores, e serve como um excelente exemplo de projeto de computadores CISC. O PowerPC é um descendente direto do primeiro sistema RISC, o IBM 801, e é um dos mais poderosos e bem projetados sistemas RISC existentes no mercado.

Nesta seção, apresentamos uma descrição sucinta desses dois sistemas.

Pentium

A Intel permanece, há várias décadas, como o maior fabricante de microprocessadores. A evolução dos seus microprocessadores constitui um bom indicador da evolução da tecnologia de computadores de modo geral.

A Tabela 2.6 mostra essa evolução. Curiosamente, à medida que os microprocessadores se tornavam mais rápidos e complexos, a Intel acompanhava cada passo. Inicialmente, ela costumava desenvolver um novo microprocessador a cada quatro anos. Para manter seus concorrentes em segundo plano, nas três últimas gerações do Pentium, esse tempo de desenvolvimento foi reduzido de um ou dois anos.

Vale a pena apresentar alguns destaques na evolução dos processadores da Intel:

- **8080:** o primeiro microprocessador de propósito geral fabricado no mundo. Tratava-se de um processador de 8 bits, com transferência de dados à memória de 8 bits. Foi utilizado no primeiro computador pessoal, o Altair.
- **8086:** um processador de 16 bits bem mais poderoso. Além de registradores e barramento com maior número de bits, o 8086 possuía uma memória cache, ou fila, de instruções que buscava antecipadamente algumas instruções na memória antes de elas serem executadas. Uma variante desse processador, o 8088, foi usado no primeiro computador pessoal da IBM, garantindo o sucesso da Intel.
- **80286:** uma extensão do 8086, que possibilitava endereçar uma memória de 16 Mbytes, em vez de apenas 1 Mbyte.
- **80386:** foi o primeiro microprocessador de 32 bits da Intel e representou a maior revisão da sua linha de produtos. Com uma arquitetura de 32 bits, o 80386 competia em complexidade e poder de computação com os minicomputadores e computadores de grande porte introduzidos apenas alguns anos antes. Esse foi o primeiro processador da Intel a oferecer suporte à multitarefa, ou seja, para execução simultânea de vários programas.
- **80486:** este processador introduziu uma tecnologia de cache muito mais elaborada e poderosa e uma *pipeline* de instruções sofisticada. Ele embutia também um co-processador aritmético de ponto flutuante, aliviando a CPU principal do processamento de operações aritméticas complexas.
- **Pentium:** com o Pentium, a Intel introduziu o uso da tecnologia superescalar, que permite a execução de múltiplas instruções em paralelo.
- **Pentium Pro:** deu continuidade à utilização da organização superescalar iniciada com o Pentium, com uso intensivo de renomeação de registradores, previsão de desvios, análise do fluxo de dados, assim como execução especulativa de instruções.
- **Pentium II:** incorporou a tecnologia Intel MMX, projetada especificamente para processar eficientemente vídeo, áudio e dados gráficos.
- **Pentium III:** incorpora instruções de ponto flutuante adicionais para apoiar software gráfico em 3D.
- **Merced:** essa nova geração de processadores Intel usa uma organização de 64 bits.

PowerPC

Em 1975, o projeto do minicomputador 801 da IBM antecipou muitos dos conceitos de arquitetura usados nos sistemas RISC. O 801, juntamente com o processador RISC I da Universidade da Califórnia, em Berkeley, lançou o movimento das máquinas RISC. Entretanto, ele era simplesmente um protótipo, construído com o objetivo de demonstrar conceitos de projeto. O sucesso do projeto 801 levou a IBM a desenvolver uma estação de trabalho RISC comercial, o RT PC, lançado em 1986, adaptando os conceitos da arquitetura do 801 para um produto real. O RT PC não foi um sucesso comercial e teve muitos rivais com desempenho comparável ou melhor. Em 1990, a IBM produziu um terceiro sistema, valendo-se das lições do 801 e do RT PC. O IBM RISC Sistema 6000 era uma máquina RISC superescalar, vendida como uma estação de trabalho de alto desempenho; logo após seu lançamento, a IBM passou a chamar essa arquitetura de arquitetura POWER.

Para o passo seguinte, a IBM aliou-se à Motorola, que havia desenvolvido a série de microprocessadores 68000, e à Apple, que usava a pastilha da Motorola nos seus computadores Macintosh. O resultado foi uma série de máquinas que implementam a arquitetura PowerPC. Essa arquitetura era derivada da arquitetura POWER, com pequenas alterações que visavam adicionar características importantes, possibilitar uma implementação mais eficiente, pela redução do número de instruções, e relaxar a especificação para resolver problemas em alguns casos especiais. A arquitetura PowerPC resultante constitui um sistema RISC superescalar. O PowerPC é usado em milhões de máquinas Apple Macintosh e em numerosas aplicações de pastilha embutida. Um exemplo dessa última aplicação é a família de pastilhas de gerenciamento de rede da IBM, que podem ser embutidas em equipamentos de rede para dar gerenciamento de acesso de usuários, em plataformas constituídas de equipamentos de diferentes fabricantes.

Os principais membros da família PowerPC são relacionados a seguir (Tabela 2.8):

- **601:** seu objetivo foi lançar no mercado, o mais rápido possível, a arquitetura do PowerPC. O 601 é um processador de 32 bits.
- **603:** voltado para microcomputadores e computadores portáteis, também é um processador de 32 bits, com desempenho comparável ao 601, mas de menor custo e implementação mais eficiente.
- **604:** voltado para microcomputadores e máquinas servidoras de menor desempenho, é também um processador de 32 bits, mas utiliza muito mais as técnicas avançadas de projeto de processadores superescalares para obter maior desempenho.
- **620:** voltado para máquinas servidoras de alto desempenho foi o primeiro membro da família PowerPC a implementar uma arquitetura completa de 64 bits, com registradores e barramento de dados de 64 bits.
- **740/750:** também conhecido como processador G3, integra dois níveis de memória cache na pastilha do processador principal, o que resulta em uma melhora de desempenho significativa em relação a máquinas com memória cache fora da pastilha.
- **G4:** esse processador fornece ainda maior paralelismo e velocidade interna da pastilha do processador.

Tabela 2.8 Resumo do processador PowerPC

	601	603/603e	604/604e	740/750 (G3)	G4
Data de lançamento	1993	1994	1994	1997	1999
Velocidade de relógio (MHz)	50 – 120	100 – 300	166 – 350	200 – 366	500
Cache L1	—	16 Kb instr. 16 Kb dados	32 Kb instr. 32 Kb dados	32 Kb instr. 32 Kb dados	32 Kb instr. 32 Kb dados
Cache secundária L2	—	—	—	256 Kb – 1 Mb	256 Kb – 1 Mb
Número de transistores (10^6)	2,8	1,6 – 2,6	3,6 – 5,1	6,35	

2.4 LEITURA E SITES WEB RECOMENDADOS

Uma descrição da série 7000 da IBM pode ser encontrada em Bell e Newell (1971a). Existem boas descrições do IBM 360 em Siewiorek e outros (1982) e do PDP-8 e de outras máquinas DEC em Bell e outros (1978a). Esses três livros também contêm exemplos detalhados de vários outros computadores, incluindo a história dos computadores até o início dos anos 80. Um livro mais recente, que apresenta um excelente conjunto de estudos de caso relativos a máquinas históricas, é de autoria de Blaauw e Brooks (1997). Uma boa referência para a história dos microprocessadores é a obra de Betker e outros (1997).

Uma das melhores descrições do Pentium é apresentada em Shanley (1998). A própria documentação da Intel (Intel, 1998; e Intel, 1998b) é também muito boa. Brey (1997) fornece uma visão completa da linha de microprocessadores Intel, com ênfase nas máquinas de 32 bits.

Um cuidadoso tratamento da arquitetura PowerPC pode ser encontrado em IBM (1994). Shanley (1995a) faz uma exposição análoga, além de uma descrição do 601, e Weiss e Smith (1994) tratam das arquiteturas POWER e PowerPC.

Para interessantes discussões sobre a Lei de Moore e suas conseqüências, veja Hutcheson e Hutcheson (1996), Schaller (1997) e Bohr (1998).



Sites Web recomendados:

- **Charles Babbage Institute:** contém endereços de vários sites Web sobre a história de computadores.
- **PowerPC:** site Web da IBM sobre o PowerPC.
- **Developer Home:** site Web da Intel para desenvolvedores de sistemas; uma referência inicial com informações sobre o Pentium.

2.5 EXERCÍCIOS

2.1 Considere o problema de somar dois vetores (unidimensionais) $A = A(1), A(2), ..., A(1000)$ e $B = B(1), B(2) ... B$ (1000), cada um com mil números, para obter um vetor C, tal que $C(I) = A(I) + B(I)$ para $I = 1, 2, ..., 1000$. Escreva um programa para resolver esse problema usando o conjunto de instruções do IAS.

2.2 Nos modelos 65 e 75 do IBM 360, os endereços são divididos em duas unidades de memória distintas (por exemplo, todas as palavras de endereço par são alocadas em uma unidade e todas as palavras de endereço ímpar na outra). Qual seria a finalidade dessa técnica?

PARTE

2

O SISTEMA DE COMPUTAÇÃO

OBJETIVOS

Um sistema de computação consiste em um processador, memória, dispositivos de E/S e interconexões entre esses componentes principais. A Parte II aborda cada um desses componentes detalhadamente, com exceção do processador, que em razão de sua complexidade será tratado minuciosamente na Parte III.

ROTEIRO

Capítulo 3: Barramentos do sistema

No nível mais alto, um sistema de computação pode ser descrito a partir da funcionalidade de cada um dos seus componentes principais, da estrutura de interconexão e do tipo de sinais trocados entre esses componentes. O capítulo aborda a estrutura de interconexão e as trocas de sinais por meio dessa estrutura. Um enfoque maior será dado ao mecanismo de interconexão mais comum: o barramento ou conjunto de barramentos. Este capítulo trata também dos principais aspectos do projeto de mecanismos de interconexão, particularmente a necessidade de suportar interrupções.

Capítulo 4: Memória interna

Este capítulo trata da organização da memória principal e do uso de memórias cache para obtenção de melhor desempenho. O projeto de um sistema de memória principal deve levar em conta três aspectos conflitantes: os requisitos de maior capacidade de armazenamento, de tempo de acesso mais rápido e de custo mais baixo. À medida que a tecnologia de memória evolui, os valores anteriormente válidos para esses parâmetros vão sendo alterados e, portanto, em cada nova implementação, as decisões de projeto relativas à organização da memória principal devem ser revistas. Duas áreas de particular interesse são enfatizadas neste capítulo: a organização das memórias cache e os vários esquemas de memórias dinâmicas de acesso aleatório (*dynamic random-access memory* — DRAM).

Capítulo 5: Memória externa

O armazenamento permanente de grandes quantidades de dados requer uma organização de memória externa. O tipo de memória externa mais utilizado é o disco magnético, abordado na maior parte do capítulo. Primeiramente, discutimos a tecnologia de discos magnéticos e alguns aspectos do projeto. Em seguida, examinamos como o uso da organização RAID pode resultar em melhor desempenho da memória de discos. Esse capítulo trata ainda do armazenamento de dados em fitas e discos ópticos.

Capítulo 6: Entrada e saída

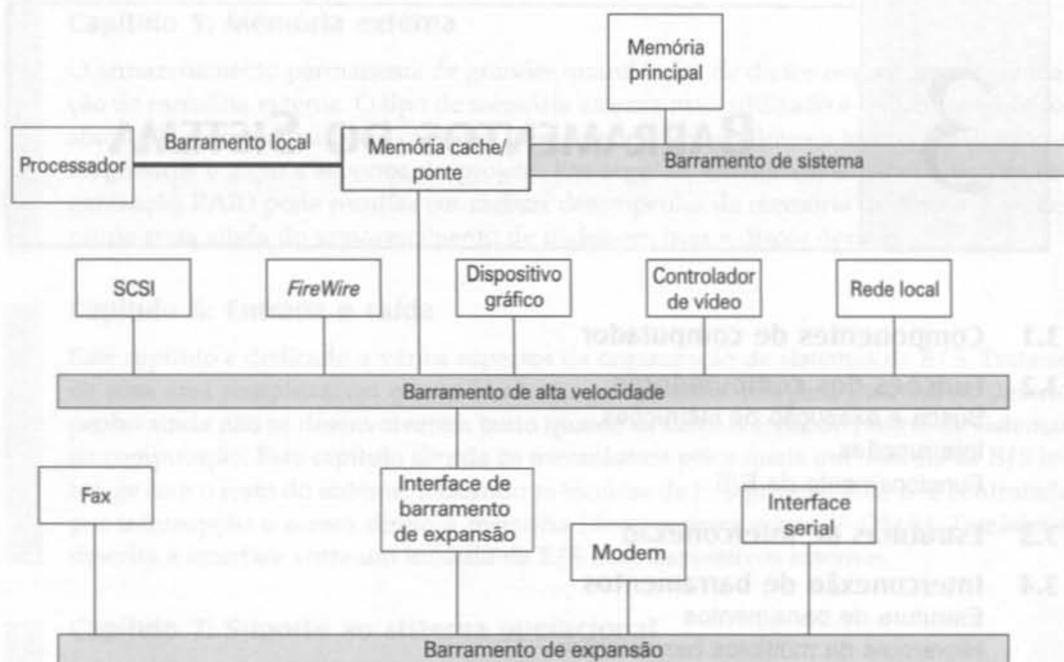
Este capítulo é dedicado a vários aspectos da organização de sistemas de E/S. Trata-se de uma área complexa, em que as técnicas a serem utilizadas para melhora do desempenho ainda não se desenvolveram tanto quanto as demais áreas de projeto de sistemas de computação. Este capítulo aborda os mecanismos pelos quais um módulo de E/S interage com o resto do sistema, incluindo as técnicas de E/S programada, E/S controlada por interrupção e acesso direto à memória (*direct memory access* — DMA). Também é descrita a interface entre um módulo de E/S e os dispositivos externos.

Capítulo 7: Suporte ao sistema operacional

Uma abordagem detalhada a respeito de sistemas operacionais foge ao escopo deste livro. É importante, entretanto, entender as funções básicas de um sistema operacional e saber como ele interage com o hardware do sistema para conseguir um bom desempenho. Este capítulo descreve os princípios básicos de sistemas operacionais e discute as características específicas do projeto no hardware do computador para dar suporte ao sistema operacional.

capítulo

3 Barramentos do Sistema



- Um ciclo de execução de uma instrução é formado pela seguinte seqüência de operações: busca da instrução, seguida de zero ou mais operações de busca de operandos, de zero ou mais operações de armazenamento de dados, de teste de interrupção (caso as interrupções estejam habilitadas).
- Os componentes principais de um computador (processador, memória principal, módulos de E/S) precisam ser conectados entre si, para que possam trocar dados e sinais de controle. O mecanismo mais comum de interconexão usa um barramento do sistema compartilhado com múltiplas linhas. Os sistemas mais modernos usam uma hierarquia de barramentos para obter melhor desempenho.
- Os principais aspectos de projeto de um sistema de barramentos são a arbitração (a decisão sobre permissões para envio de sinais por meio das linhas do barramento pode ser centralizada ou distribuída), a temporização (o envio de sinais por meio dos barramentos pode ser sincronizado por um relógio central ou pode ser feito de maneira assíncrona, com base na transmissão mais recente) e a largura do barramento (o número de linhas de endereço e de linhas de dados).

No nível mais alto, um computador é composto pela CPU, memória e dispositivos de E/S, podendo conter um ou mais de cada um desses componentes. Para desempenhar a função básica de um computador, ou seja, executar programas, esses componentes devem ser conectados de alguma maneira. Desse modo, para mostrar o funcionamento de um sistema de computação nesse nível mais alto, devemos descrever (1) o comportamento externo de cada componente — os dados e os sinais de controle que ele troca com os demais componentes — e (2) a estrutura da interconexão.

Essa visão global da estrutura e do funcionamento de um computador é importante para que se possa entender sua natureza, além de questões cada vez mais complexas, relativas à avaliação de seu desempenho. Uma boa compreensão da estrutura e do funcionamento de um computador permite identificar pontos críticos para o desempenho do sistema, visualizar caminhos alternativos, prever efeitos de falhas no sistema, em caso de falha de componentes, e avaliar a dificuldade em introduzir melhorias de desempenho. Em muitos casos, requisitos de maior capacidade/desempenho e de maior tolerância a falhas são atendidos mediante mudanças no projeto e não simplesmente por meio do aumento da velocidade e da confiabilidade de componentes individuais.

Este capítulo aborda as estruturas básicas de interconexão dos componentes de um computador. Para fornecer os conceitos fundamentais necessários, ele inicia com um breve exame dos componentes básicos de um sistema de computação e seus requisitos de interface. Em seguida, é apresentada uma visão funcional do sistema.

Estaremos então preparados para examinar o uso de barramentos como mecanismo de interconexão dos componentes do sistema.

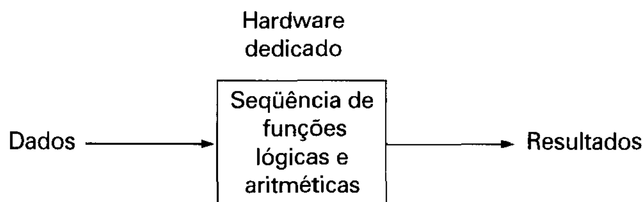
3.1 COMPONENTES DE COMPUTADOR

Como foi discutido no Capítulo 2, praticamente todos os projetos de computadores atuais são fundamentados nos conceitos desenvolvidos por John von Neumann no Instituto de Estudos Avançados de Princeton. Esse projeto, conhecido como *Arquitetura von Neumann*, é baseado em três conceitos básicos:

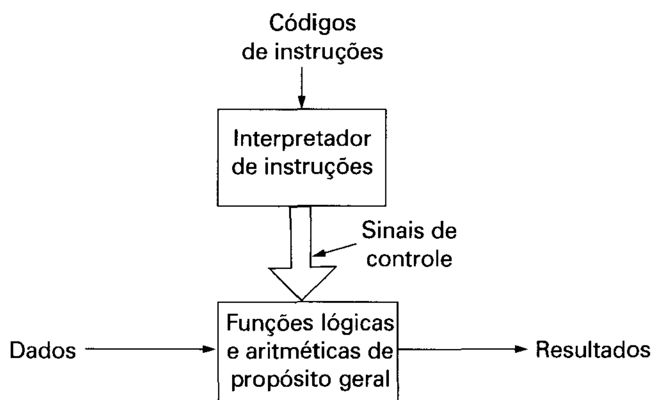
- Os dados e as instruções são armazenados em uma única memória de leitura e escrita.
- O conteúdo dessa memória é endereçado pela sua posição, independentemente do tipo de dados nela contidos.
- A execução de instruções ocorre de modo seqüencial (exceto quando essa seqüência é explicitamente alterada de uma instrução para a seguinte).

Embora as idéias básicas por trás desses conceitos já tenham sido discutidas no Capítulo 1, vale a pena resumi-las aqui. O computador é composto de um pequeno conjunto de componentes lógicos básicos, que podem ser combinados de vários modos para armazenar dados binários e executar operações aritméticas e lógicas sobre esses dados. É possível obter, para cada aplicação particular, uma configuração de componentes lógicos projetada especificamente para executar essa aplicação. Esse processo de conectar os diferentes componentes do sistema para obter a configuração desejada pode ser concebido como uma forma de programação. O 'programa' resultante é formado pelo hardware e é chamado *programa hardwired*.

Considere agora outra alternativa: suponha que construímos uma configuração de funções lógicas e aritméticas de propósito geral. Esse conjunto de componentes de hardware é capaz de executar várias funções sobre os dados, dependendo dos sinais de controle que lhe são aplicados. Na situação anterior, em que o hardware é dedicado para uma aplicação particular, o sistema apenas lê dados e produz resultados (Figura 3.1a). Um hardware de propósito geral é capaz de ler dados e sinais de controle e produzir resultados. Assim, em vez de projetar um novo hardware para cada aplicação nova, o programador simplesmente precisa fornecer um novo conjunto de sinais de controle.



(a) Programação em hardware



(b) Programação em software

Figura 3.1 Abordagens de programação em hardware e em software.

Como esses sinais de controle devem ser fornecidos? A resposta é simples, porém sutil. Um programa é constituído de uma seqüência de passos. A cada passo, alguma operação lógica ou aritmética é executada sobre algum dado. Para cada passo, é necessário um novo conjunto de sinais de controle. Podemos definir um código para cada possível conjunto de sinais de controle e acrescentar ao hardware de propósito geral um elemento capaz de interpretar esses códigos e gerar os sinais de controle correspondentes (Figura 3.1b).

Programar agora ficou muito mais fácil. Em vez de projetar um novo hardware para cada aplicação nova, precisamos apenas fornecer uma nova seqüência de códigos. Cada código corresponde a uma instrução; uma parte do hardware interpreta essas instruções e gera os sinais de controle correspondentes. Para distinguir esse novo método de programação, uma seqüência de códigos ou instruções é chamada de *software*.

A Figura 3.1b indica os dois componentes mais importantes do sistema: o módulo que interpreta as instruções e o módulo que executa as funções lógicas e aritméticas de propósito geral. Esses componentes constituem a CPU. Vários outros componentes são necessários para que um computador possa funcionar. Os dados e as instruções devem ser introduzidos no sistema de alguma maneira. Para isso, é necessário um módulo de entrada de dados. Esse módulo contém componentes básicos que recebem dados e instruções, em algum formato, e os converte em uma representação interna, composta de sinais usados pelo sistema. Além disso, o sistema deve ser capaz de mostrar os resultados produzidos; isso é feito por um módulo de saída de dados. Esses dois módulos são denominados *componentes de E/S*.

É necessário mais um componente. Um dispositivo de entrada fornece as instruções e os dados seqüencialmente. No entanto, um programa nem sempre é executado seqüencialmente, podendo incluir desvios (por exemplo, a instrução *jump* do IAS). Do mesmo modo, uma operação pode precisar acessar mais de um dado de cada vez, em uma seqüência predeterminada. Dessa maneira, deve existir um local para armazenar instruções e dados temporariamente. Esse módulo é chamado *memória*, ou *memória principal*, para distingui-la da área de armazenamento externo ou dispositivos periféricos. Von Neumann mostrou que a mesma memória poderia ser usada para armazenar tanto instruções quanto dados.

A Figura 3.2 ilustra esses componentes de alto nível do sistema e sugere as interações entre eles. A CPU troca dados com a memória. Para isso, ela tipicamente usa dois registradores internos da CPU: um registrador de endereçamento à memória (*memory address register* — MAR), que especifica o endereço da memória a ser usado pela próxima instrução de leitura ou escrita, e um registrador temporário de dados (*memory buffer register* — MBR), que contém um valor a ser gravado na memória ou recebe um valor lido da memória. Da mesma maneira, o registrador de endereçamento de E/S (*I/O address register* — I/O AR) especifica um determinado dispositivo de E/S. Um registrador temporário de dados de E/S (I/O buffer register — I/O BR) é usado para a troca de dados entre um módulo de E/S e a CPU.

Um módulo de memória é constituído de um conjunto de posições de memória identificadas por endereços numerados seqüencialmente. Cada posição contém um número binário que pode ser interpretado como uma instrução ou como um dado. Um módulo de E/S transfere dados de dispositivos externos para a CPU e para a memória e vice-versa. Ele contém áreas de armazenamento temporário (*buffers*) internas, para guardar temporariamente esses dados até que possam ser enviados.

Com uma visão desses componentes principais, podemos agora voltar nossa atenção para o funcionamento desses componentes para a execução de programas.

3.2 FUNÇÕES DOS COMPUTADORES

A função básica desempenhada por um computador é executar um programa que é constituído por um conjunto de instruções armazenadas na memória. O processador realiza o trabalho efetivo de executar as instruções especificadas no programa. Esta seção apresenta uma visão geral dos principais elementos da execução de programas. Em sua forma mais simples, há dois passos para o processamento de instruções: o processador lê (*busca*) instruções na memória, uma de cada vez, e executa cada uma. A execução de um programa consiste na repetição desse processo de busca e execução de instruções. A execução de uma instrução pode envolver diversas operações, dependendo da natureza da instrução (veja, por exemplo, a parte inferior da Figura 2.4).

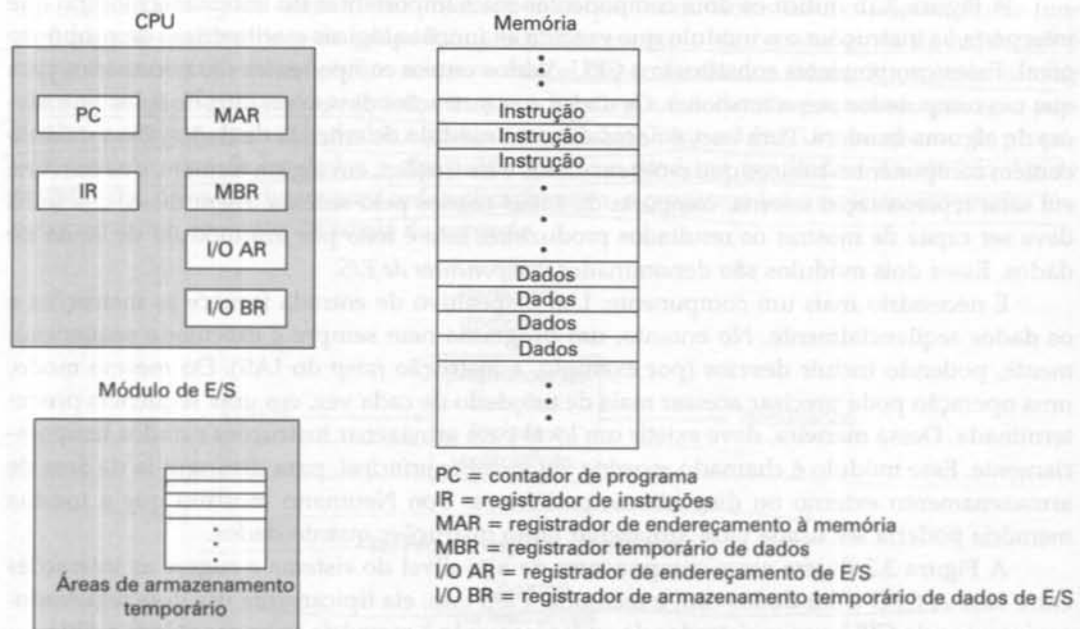


Figura 3.2 Componentes do computador: visão global.

O processamento necessário para a execução de uma instrução é chamado de *ciclo de instrução*. Usando nossa descrição simplificada de dois passos, o ciclo de instrução é mostrado na Figura 3.3. Os dois passos são denominados *ciclo de busca* e *ciclo de execução*. A execução de programas encerra-se somente se a máquina for desligada, se ocorrer algum tipo de erro irrecuperável ou se for executada uma instrução de programa que pare a operação do computador.

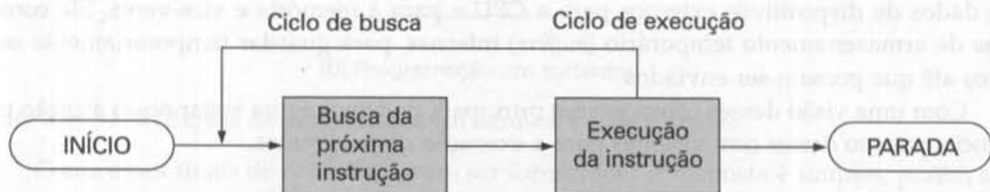


Figura 3.3 Ciclo de instrução básico.

Busca e execução de instruções

No início de cada ciclo de instrução, o processador busca uma instrução da memória. Em um processador típico, um registrador chamado *contador de instruções* ou 'contador de programa' (*program counter* — PC) é usado para guardar o endereço da próxima instrução a ser buscada na memória. Normalmente, o processador sempre incrementa o PC depois de cada busca de instrução, de modo que a próxima instrução esteja em uma seqüência (isto é, ela está localizada no endereço de memória seguinte). Por exemplo, considere um computador no qual cada palavra de memória tem 16 bits. Suponha que o PC contenha o endereço 300. O

processador buscará a próxima instrução na posição de memória de endereço 300. Nos ciclos de instruções seguintes, ele buscará as instruções armazenadas nas posições de memória de endereços 301, 302, 303, e assim por diante. Essa seqüência pode ser alterada, como explicamos a seguir.

A instrução buscada na memória é carregada no registrador do processador conhecido como registrador de instruções (*instruction register* — IR). Ela contém bits que especificam a ação que o processador deve executar. O processador interpreta a instrução e executa a ação requisitada. Em geral, essas ações são classificadas em quatro categorias:

- **Processador-memória:** transferência de dados do processador para a memória ou da memória para o processador.
- **Processador-E/S:** transferência de dados entre o processador e um dispositivo periférico por meio de um módulo de E/S.
- **Processamento de dados:** execução de operações aritméticas ou lógicas sobre os dados.
- **Controle:** determinadas instruções podem especificar que a seqüência de execução de instruções seja alterada. Por exemplo, o processador pode buscar uma instrução na posição de memória de endereço 149, que especifica que a próxima instrução a ser executada é aquela contida na posição de memória de endereço 182. A execução dessa instrução consiste em armazenar o endereço 182 no PC. Assim, no próximo ciclo de busca, a instrução será obtida do endereço 182, e não do endereço 150.

A execução de uma instrução pode envolver uma combinação dessas ações.

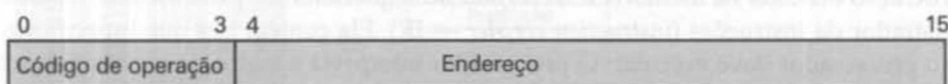
Considere um exemplo simples, que utiliza uma máquina hipotética com as características ilustradas na Figura 3.4. O processador possui apenas um registrador de armazenamento de dados, denominado acumulador (AC). Instruções e dados têm, ambos, tamanho de 16 bits. É conveniente, portanto, organizar a memória em palavras de 16 bits. O formato das instruções contém 4 bits para o código de operação, podendo existir, assim, $2^4 = 16$ códigos de operação distintos e até $2^{12} = 4.096$ (4K) palavras de memória que podem ser endereçadas diretamente.

A Figura 3.5 ilustra a execução parcial de um programa, mostrando as partes da memória e os registradores relevantes[1]. O trecho de programa ilustrado acrescenta o conteúdo da palavra de memória de endereço 940 ao conteúdo da palavra de memória de endereço 941 e armazena o resultado nesse último endereço. São necessárias três instruções, que podem ser descritas em três ciclos de busca e de execução:

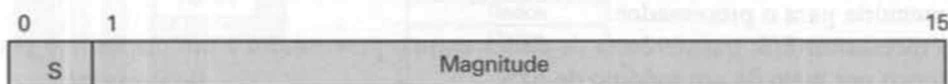
1. O conteúdo do PC é 300, o endereço da primeira instrução. Essa instrução é carregada dentro do registrador de instrução, IR. Note que esse processo envolve o uso do registrador de endereçamento à memória (MAR) e do registrador temporário de dados (MBR). Por simplicidade, esses registradores intermediários são ignorados.
2. Os quatro primeiros bits do IR indicam que um valor deve ser armazenado no registrador AC. Os 12 bits restantes especificam o endereço 940, de onde o valor deve ser obtido.
3. O PC é incrementado e a próxima instrução é buscada.

1 É usada a notação hexadecimal, em que cada dígito representa 4 bits. Essa notação é a mais conveniente para representar o conteúdo da memória e de registradores quando a palavra possui tamanho múltiplo de 4. Ela é revisada no apêndice do Capítulo 8.

4. O conteúdo de AC é somado com o conteúdo da posição de memória de endereço 941 e o resultado é armazenado em AC.
5. O PC é incrementado e a próxima instrução é buscada.
6. O conteúdo de AC é armazenado na posição de memória de endereço 941.



(a) Formato de instruções



(b) Formato de números inteiros

Contador de programa (PC) = endereço da próxima instrução

Registrador de instruções (IR) = instrução que está sendo executada

Acumulador (AC) = armazenamento temporário de dados

(c) Registradores internos da CPU

0001 = carregar AC a partir do endereço de memória especificado

0010 = armazenar o valor contido em AC no endereço de memória especificado

0101 = acrescentar ao valor contido em AC o valor contido no endereço de memória especificado

(d) Lista parcial de códigos de operação

Figura 3.4 Características de uma máquina hipotética.

Nesse exemplo, são necessários três ciclos de instruções, cada um com um ciclo de busca e um ciclo de execução, para somar os conteúdos dos endereços de memória 940 e 941. Em um computador com um conjunto de instruções mais complexo, um número menor de ciclos poderia ser necessário. Processadores modernos incluem instruções que contêm mais de um endereço. Desse modo, o ciclo de determinada instrução pode envolver mais de uma referência à memória. Uma instrução pode também especificar uma operação de E/S, em vez de uma referência à memória.

Por exemplo, a instrução ADD *B*,*A* do PDP-11 armazena a soma dos valores contidos nas posições de memória com endereços *B* e *A* no *A*. Apenas um único ciclo de instrução é executado com os seguintes passos:

- A instrução ADD é buscada.
- O conteúdo do endereço de memória *A* é carregado no processador.

- O conteúdo do endereço de memória *B* é carregado no processador. Para não perder o conteúdo de *A*, o processador deve ter no mínimo dois registradores de dados para armazenar esses valores, em vez de um único acumulador.
- Os dois valores são somados.
- O resultado obtido é armazenado no endereço de memória *A*.

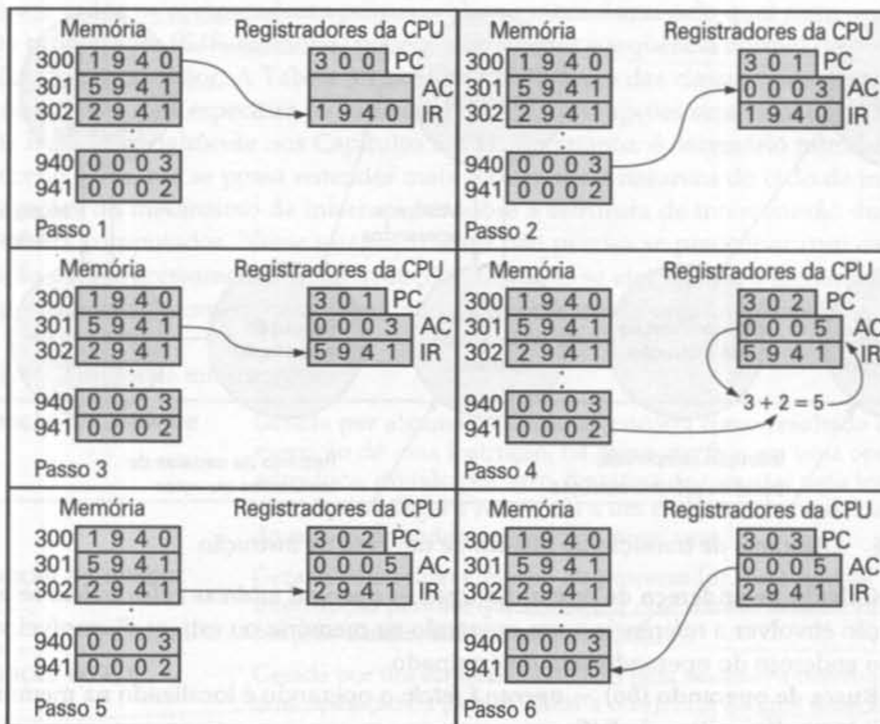


Figura 3.5 Exemplo ilustrativo de execução de um trecho de programa.

Portanto, o ciclo de execução de uma dada instrução pode envolver mais de uma referência à memória. A instrução pode também especificar uma operação de E/S, em vez de uma referência à memória. Levando-se em conta essas considerações adicionais, a Figura 3.6 fornece uma visão mais detalhada do ciclo básico de instrução mostrado na Figura 3.3. Ela é apresentada na forma de um diagrama de transição de estados. Para um dado ciclo de instrução, alguns estados podem ser nulos e outros podem ser visitados mais de uma vez. Os estados podem ser descritos como:

- **Cálculo de endereço de instrução (*cei*) — *instruction address calculation*:** o endereço da próxima instrução a ser executada é determinado. Isso geralmente envolve a soma de um valor constante ao endereço da instrução anterior. Por exemplo, se cada instrução tem um tamanho de 16 bits e a memória está organizada em palavras de 16 bits, então é adicionado o valor 1 ao endereço anterior. No entanto, se a memória estiver organizada de modo que endereça individualmente bytes de 8 bits, somam-se 2 ao endereço anterior.

- **Busca de instrução (*bi*) — *instruction fetch*:** uma instrução é lida da memória e armazenada no processador.
- **Decodificação de instrução (*di*) — *instruction operation decoding*:** o código da instrução a ser executada é analisado, para determinar qual é a operação a ser realizada e o(s) operando(s) a ser(em) usado(s).

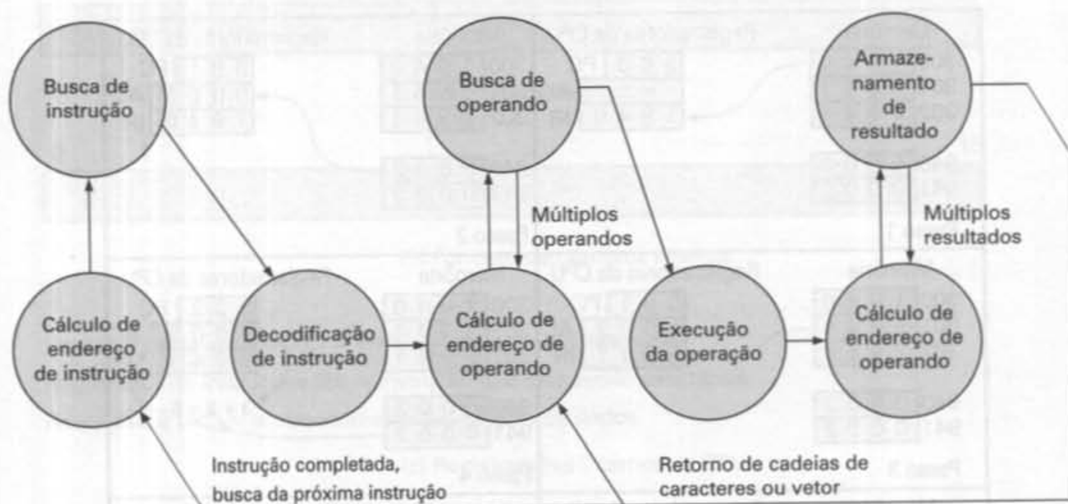


Figura 3.6 Diagrama de transição de estados de um ciclo de instrução.

- **Cálculo de endereço de operando (*ceo*) — *operand address calculation*:** se a operação envolver a referência a um operando na memória ou estiver disponível via E/S, o endereço do operando será determinado.
- **Busca de operando (*bo*) — *operand fetch*:** o operando é localizado na memória ou é lido no dispositivo de E/S.
- **Execução da operação (*eo*) — *data operation*:** a operação indicada na instrução é executada.
- **Armazenamento de resultado (*ar*) — *operand store*:** o resultado é escrito na memória ou no dispositivo de E/S.

Os estados na parte superior da Figura 3.6 envolvem transferências de valores entre o processador, de um lado, e a memória ou um dispositivo de E/S, de outro. Os estados na parte inferior do diagrama envolvem apenas operações realizadas internamente no processador. O estado *ceo* aparece duas vezes, pois uma instrução pode envolver uma operação de leitura, de escrita ou ambas. Entretanto, a ação executada nesse estado é essencialmente a mesma nos dois casos e, portanto, um único código é necessário para identificar o estado.

Note também que o diagrama possibilita representar ciclos de instruções com múltiplos operandos e múltiplos resultados, como ocorre em algumas máquinas. Por exemplo, a execução da instrução ADD *A,B* do PDP-11 obedece à seguinte seqüência de estados: *cei, bi, di, ceo, bo, ceo, bo, eo, ceo, ar*.

Finalmente, em algumas máquinas uma única instrução pode especificar uma operação a ser executada sobre um vetor de números ou sobre uma cadeia de caracteres. Como indicado na Figura 3.6, a execução dessa instrução envolve repetidas operações de busca de operando e/ou de armazenamento de resultados.

Interrupções

Quase todos os computadores possuem algum mecanismo pelo qual componentes distintos do processador (E/S, memória) podem interromper a seqüência normal de execução de instruções do processador. A Tabela 3.1 contém uma relação das classes mais comuns de interrupções. A natureza específica de cada uma dessas interrupções será examinada no decorrer deste livro, especialmente nos Capítulos 6 e 11. Entretanto, é necessário introduzir agora esse conceito, para que se possa entender mais claramente a natureza do ciclo de instrução e as implicações do mecanismo de interrupções sobre a estrutura de interconexão dos componentes de um computador. Nesse estágio, o leitor não precisa se preocupar com os detalhes da geração e do processamento de interrupções, devendo se ater apenas à comunicação entre os componentes que acontece como resultado da ocorrência de uma interrupção.

Tabela 3.1 Classes de interrupções

Interrupção de software	Gerada por alguma condição que ocorra como resultado da execução de uma instrução, tal como *overflow* em uma operação aritmética, divisão por zero, tentativa de executar uma instrução de máquina ilegal e referência a um endereço de memória fora do espaço de endereçamento do programa.
Interrupção de relógio	Gerada pelo relógio interno do processador. Esse tipo de interrupção permite que o sistema operacional execute certas funções a intervalos de tempo regulares.
Interrupção de E/S	Gerada por um controlador de E/S para sinalizar a conclusão de uma operação ou para sinalizar a ocorrência de uma situação de erro.
Interrupção de falha de hardware	Gerada na ocorrência de uma falha, tal como queda de energia ou erro de paridade na memória.

O mecanismo de interrupções visa, principalmente, melhorar a eficiência de processamento. Por exemplo, suponha que o processador esteja transferindo dados para uma impressora utilizando o esquema de ciclo de instrução da Figura 3.3. Como a maioria dos dispositivos externos é muito mais lenta que o processador, o processador deve esperar o término da operação de escrita, permanecendo ocioso até que a impressora termine de capturar os dados. Esse tempo de espera pode ser da ordem de muitas centenas ou mesmo milhares de ciclos de instrução que não envolvem acesso à memória. Isso constitui, certamente, um desperdício de tempo de processamento.

Isso é ilustrado na Figura 3.7a. O programa de usuário efetua uma série de chamadas à rotina WRITE, intercaladas com processamento. Os segmentos de código 1, 2 e 3 referem-se às seqüências de instruções que não envolvem E/S. As chamadas à rotina WRITE iniciam a execução de uma rotina do sistema que efetua a operação de E/S requisitada. Essa rotina de E/S consiste em três partes. Veja a seguir.

- Uma seqüência de instruções (marcada com o rótulo 4 na figura) para preparar a operação de E/S requisitada. Isso pode incluir a cópia de dados de saída em uma área de armazenamento temporário especial e a preparação de parâmetros para um comando de dispositivo.
- O comando de E/S propriamente dito. Se não for usado o mecanismo de interrupções, uma vez emitido esse comando, o programa deve aguardar o término da operação de E/S requisitada. Ele realiza essa espera simplesmente efetuando repetidos testes para determinar se a operação de E/S foi concluída.
- Uma seqüência de instruções (marcada com o rótulo 5 na figura) para completar a operação. Isso pode incluir a ativação de um sinal (*flag*) para indicar sucesso ou falha da operação.

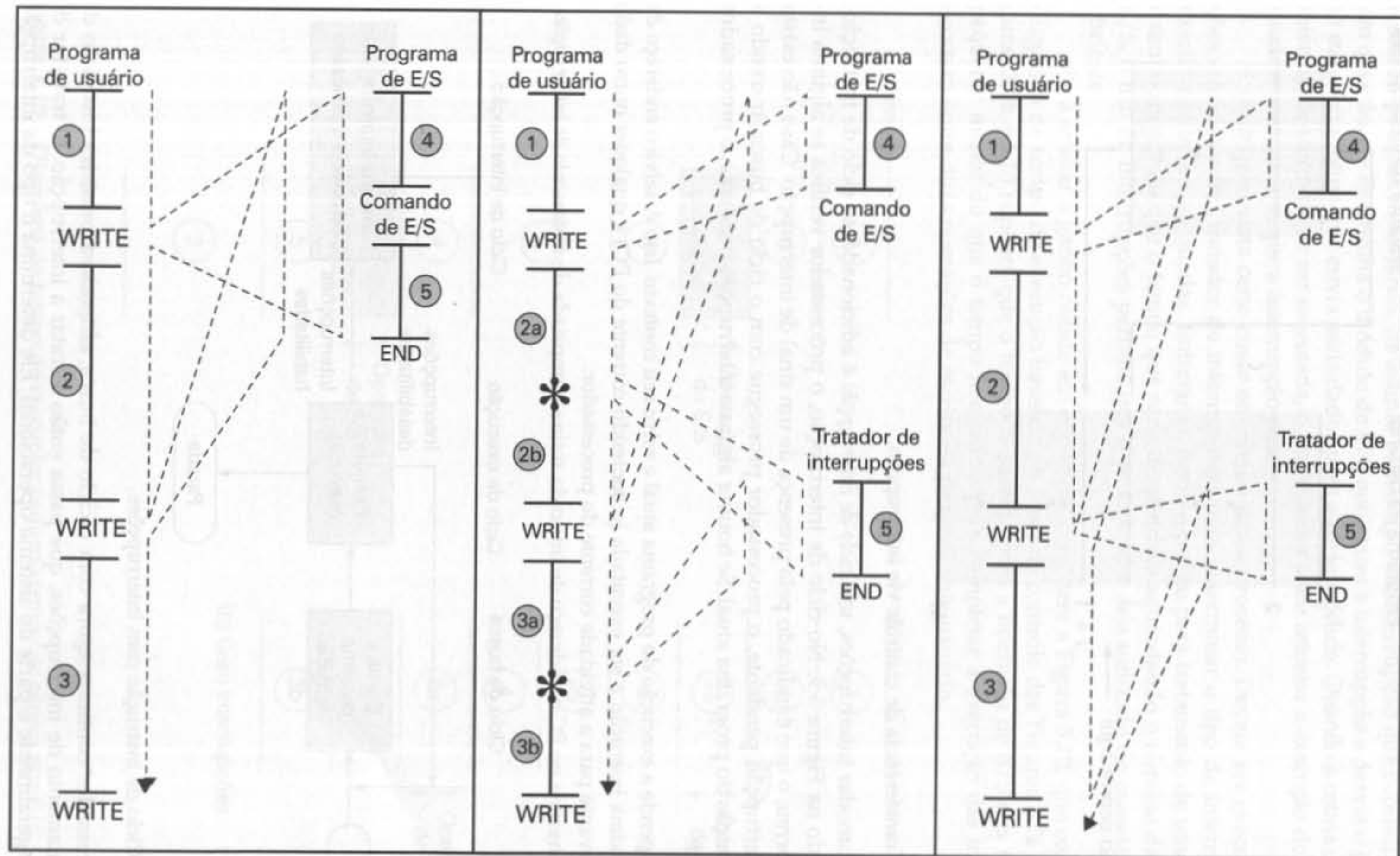
Como a operação de E/S pode levar um tempo relativamente longo para ser concluída, a execução da rotina de E/S fica suspensa, esperando pelo término da operação; por isso, o programa de usuário permanece parado no ponto de chamada à rotina WRITE por um período de tempo considerável.

As interrupções e o ciclo de instrução

Com o uso do mecanismo de interrupções, o processador pode executar outras tarefas enquanto uma operação de E/S está em andamento. Considere o fluxo de controle apresentado na Figura 3.7b. Como antes, o programa de usuário faz uma chamada de sistema na forma de uma chamada à rotina WRITE. O programa de E/S solicitado, nesse caso, consiste apenas no código de preparação e no comando de E/S requisitado. Após a execução dessas instruções, o controle retorna ao programa de usuário. Enquanto isso, o dispositivo externo se ocupa de obter os dados da memória do computador e imprimi-los. Essa operação de E/S é conduzida simultaneamente com a execução das instruções do programa de usuário.

Quando o dispositivo externo estiver pronto para ser utilizado — isto é, quando estiver pronto para receber mais dados do processador —, um sinal de *requisição de interrupção* é enviado para o processador pelo módulo de E/S desse dispositivo. Em resposta a esse sinal, o processador suspende a execução do programa atual, desviando o controle para uma rotina que controla a operação daquele dispositivo de E/S (conhecida como tratador de interrupções) e retomando a execução original após os dados terem sido enviados ao dispositivo. Os pontos em que essas interrupções ocorrem são indicados na Figura 3.7b por um asterisco.

Do ponto de vista do programa de usuário, uma interrupção é apenas isto: uma interrupção da seqüência normal de execução, que depois prossegue normalmente, quando o processamento dessa interrupção é concluído (Figura 3.8). Assim, o programa de usuário não precisa de nenhum código especial que considere a possibilidade de ele ser interrompido; o processador e o sistema operacional são responsáveis por suspender o programa de usuário e depois por retomar sua execução no mesmo ponto.



(a) Sem interrupções

(b) Interrupções; pequeno tempo de espera por E/S

(c) Interrupções; longo tempo de espera por E/S

Figura 3.7 Fluxo de controle de um programa com e sem interrupções.

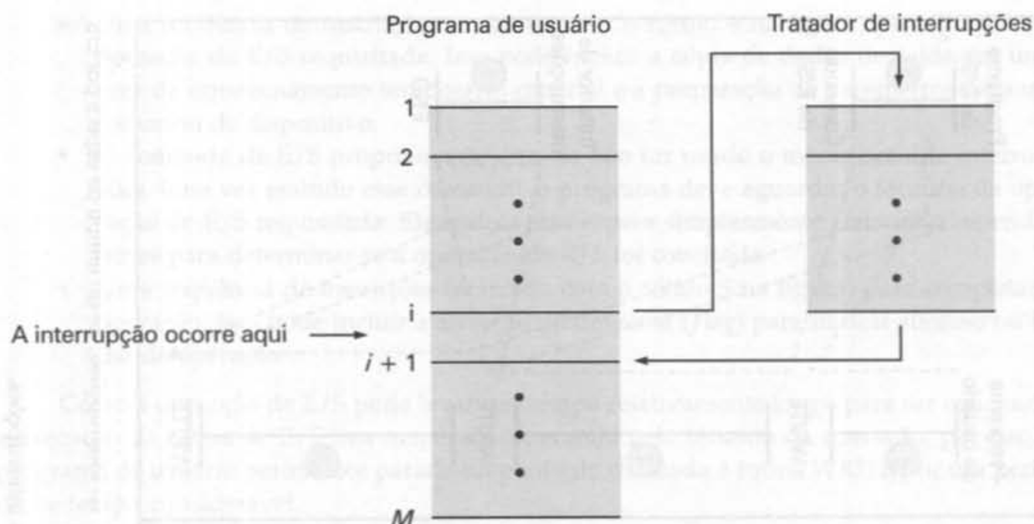


Figura 3.8 Transferência de controle via interrupção.

Para acomodar interrupções, um *ciclo de interrupção* é adicionado ao ciclo de instrução, como mostrado na Figura 3.9. No ciclo de interrupção, o processador verifica se alguma interrupção ocorreu, o que é indicado pela presença de um sinal de interrupção. Caso não exista nenhuma interrupção pendente, o processador prossegue com o ciclo de busca, trazendo a próxima instrução do programa atual. Se houver alguma interrupção pendente, o processador faz o seguinte:

1. Suspende a execução do programa atual e salva seu contexto. Isto é, salva o endereço da próxima instrução a ser executada (o conteúdo corrente do PC) e qualquer outro dado relevante para a atividade corrente do processador.
2. Armazena no PC o endereço de início da rotina apropriada de *tratamento de interrupções*.

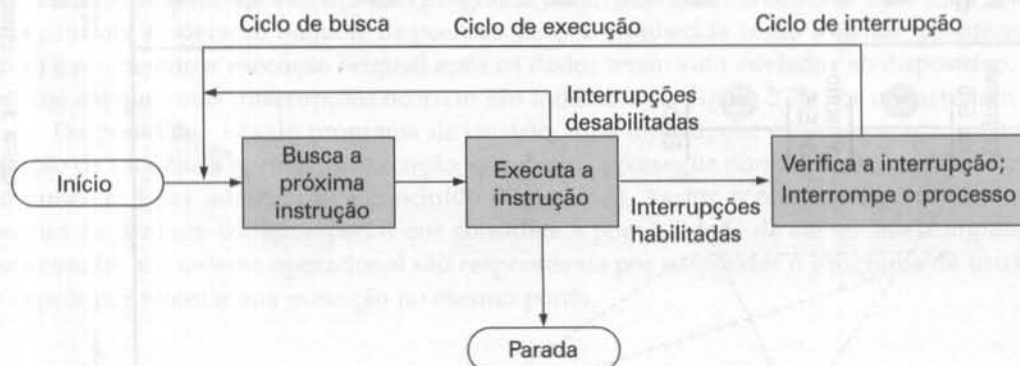


Figura 3.9 Ciclo de instrução com interrupções.

O processador continua agora com o ciclo de busca, obtendo a primeira instrução da rotina de tratamento de interrupções, que passa então a tratar a interrupção. O tratador de interrupções geralmente é parte do sistema operacional. Ele determina o tipo da interrupção

e realiza as ações necessárias, de acordo com o tipo da interrupção que ocorreu. No exemplo em questão, ele determina o módulo de E/S que gerou a interrupção e desvia o processamento para uma rotina que envia mais dados para aquele módulo. Quando a rotina do tratador de interrupções termina de ser executada, o processador pode retomar a execução do programa de usuário no ponto em que a interrupção ocorreu.

É claro que existe certo custo envolvido nesse processo. Devem ser executadas instruções adicionais (no tratador de interrupções) para determinar o tipo da interrupção e para executar as ações adequadas. Entretanto, como o tempo para tratamento de uma interrupção é muito menor do que o tempo que seria despendido aguardando o término da operação de E/S, o uso de interrupções permite que o processador seja utilizado de maneira muito mais eficiente.

Para avaliar o ganho obtido em eficiência, considere a Figura 3.10, que consiste em um diagrama de tempo de execução baseado no fluxo de controle das Figuras 3.7a e 3.7b. As Figuras 3.7b e 3.10 supõem que o tempo requerido para a operação de E/S seja relativamente pequeno: menor do que o tempo necessário para completar a execução das instruções que ocorrem entre duas operações de escrita no programa de usuário.

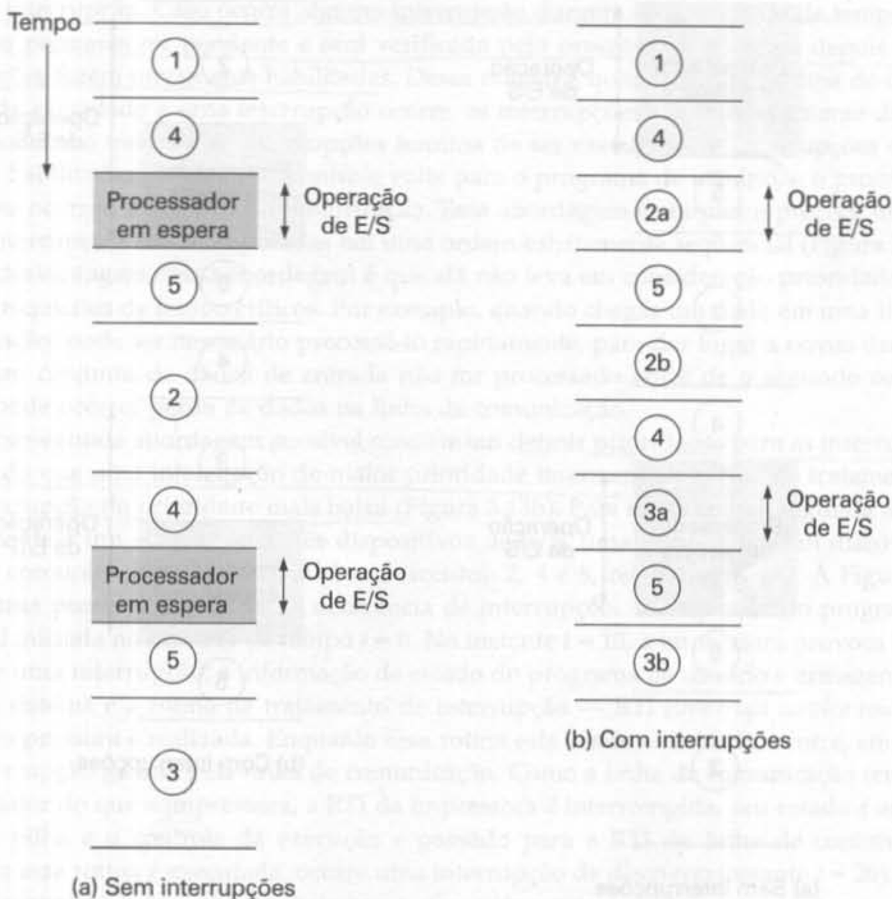


Figura 3.10 Diagrama de tempo de execução: pequeno tempo de espera por E/S.

No caso mais comum, especialmente para um dispositivo lento como uma impressora, o tempo de execução de uma operação de E/S é muito maior do que o tempo de execução de uma seqüência de instruções de um programa de usuário. Isso é mostrado na Figura 3.7c. Nesse caso, o programa de usuário executa a segunda chamada à rotina WRITE antes que a operação de E/S gerada pela primeira chamada tenha sido completada. Portanto, o programa de usuário é suspenso nesse ponto. Apenas quando a operação de E/S anterior é completada, a nova chamada à rotina WRITE pode ser processada, iniciando uma nova operação de E/S. A Figura 3.11 mostra o diagrama de tempo de execução para essa situação, com e sem o uso de interrupções. Pode-se notar que ainda há algum ganho em eficiência, pois há uma sobreposição de parte do tempo gasto na operação de E/S com a execução de instruções do programa de usuário.

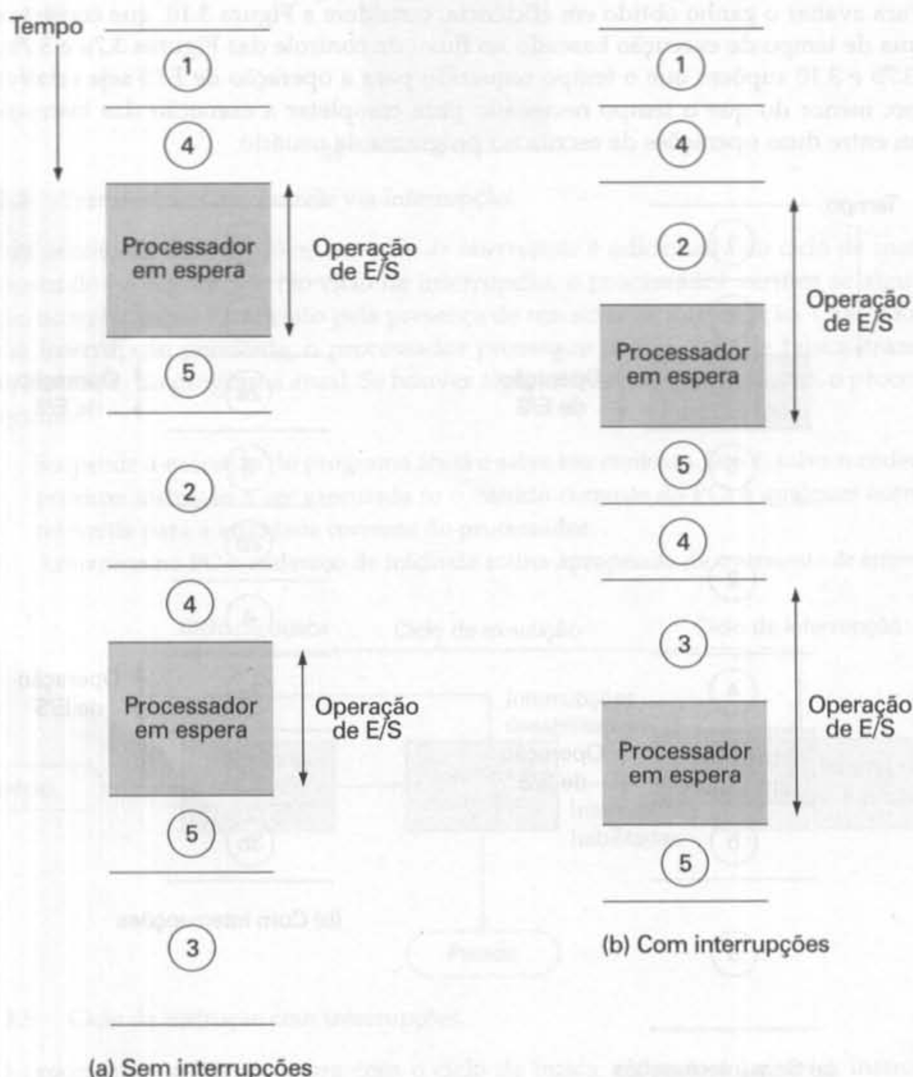


Figura 3.11 Diagrama de tempo de execução: longo tempo de espera por E/S.

A Figura 3.12 mostra um diagrama revisado de transição de estados do ciclo de instrução que inclui o processamento de ciclos de interrupção.

Múltiplas interrupções

Na discussão precedente, consideramos a possibilidade de ocorrência de uma única interrupção. Entretanto, supomos que podem ocorrer várias interrupções. Por exemplo, se um programa estiver recebendo dados de uma linha de comunicação e enviando resultados para impressão, a impressora gera uma interrupção sempre que completar uma operação de impressão e o controlador da linha de comunicação produz uma interrupção sempre que chega uma nova unidade de dados. Uma unidade de dados pode ser constituída de um único caractere ou de um bloco de caracteres, dependendo da natureza da comunicação. Em qualquer caso, pode ocorrer uma interrupção de comunicação enquanto uma interrupção da impressora está sendo processada.

Existem duas abordagens possíveis para o tratamento de múltiplas interrupções. A primeira é *desabilitar as interrupções* enquanto uma interrupção está sendo processada. Enquanto as interrupções estiverem desabilitadas, o processador pode ignorar qualquer sinal de requisição de interrupção. Caso ocorra alguma interrupção durante esse intervalo de tempo, a interrupção permanecerá pendente e será verificada pelo processador somente depois que as interrupções forem novamente habilitadas. Dessa maneira, quando um programa de usuário está sendo executado e uma interrupção ocorre, as interrupções são imediatamente desabilitadas. Quando o tratador de interrupções termina de ser executado, as interrupções são novamente habilitadas, antes que o controle volte para o programa de usuário, e o processador verifica se ocorreu alguma outra interrupção. Essa abordagem é simples e precisa, uma vez que as interrupções são manipuladas em uma ordem estritamente seqüencial (Figura 3.13a).

A desvantagem dessa abordagem é que ela não leva em consideração prioridades relativas ou requisitos de tempo críticos. Por exemplo, quando chegar um dado em uma linha de comunicação, pode ser necessário processá-lo rapidamente, para dar lugar a novos dados. Se o primeiro conjunto de dados de entrada não for processado antes de o segundo conjunto chegar, pode ocorrer perda de dados na linha de comunicação.

Uma segunda abordagem possível consiste em definir prioridades para as interrupções, permitindo que uma interrupção de maior prioridade interrompa a rotina de tratamento de uma interrupção de prioridade mais baixa (Figura 3.13b). Para mostrar essa segunda abordagem, considere um sistema com três dispositivos de E/S, uma impressora, um disco e uma linha de comunicação, com prioridades crescentes: 2, 4 e 5, respectivamente. A Figura 3.14 mostra uma possível seqüência de ocorrência de interrupções. A execução do programa de usuário é iniciada no instante de tempo $t = 0$. No instante $t = 10$, a impressora provoca a ocorrência de uma interrupção; a informação de estado do programa de usuário é armazenada na pilha do sistema e a rotina de tratamento de interrupção — RTI (*interrupt service routine* — ISR) da impressora é realizada. Enquanto essa rotina está sendo executada, ocorre, em $t = 15$, uma interrupção gerada pela linha de comunicação. Como a linha de comunicação tem prioridade maior do que a impressora, a RTI da impressora é interrompida, seu estado é armazenado na pilha e o controle da execução é passado para a RTI da linha de comunicação. Enquanto essa rotina é executada, ocorre uma interrupção de disco (no instante $t = 20$). Como essa interrupção tem prioridade mais baixa, seu tratamento tem de esperar até que o tratamento da interrupção de comunicação termine.

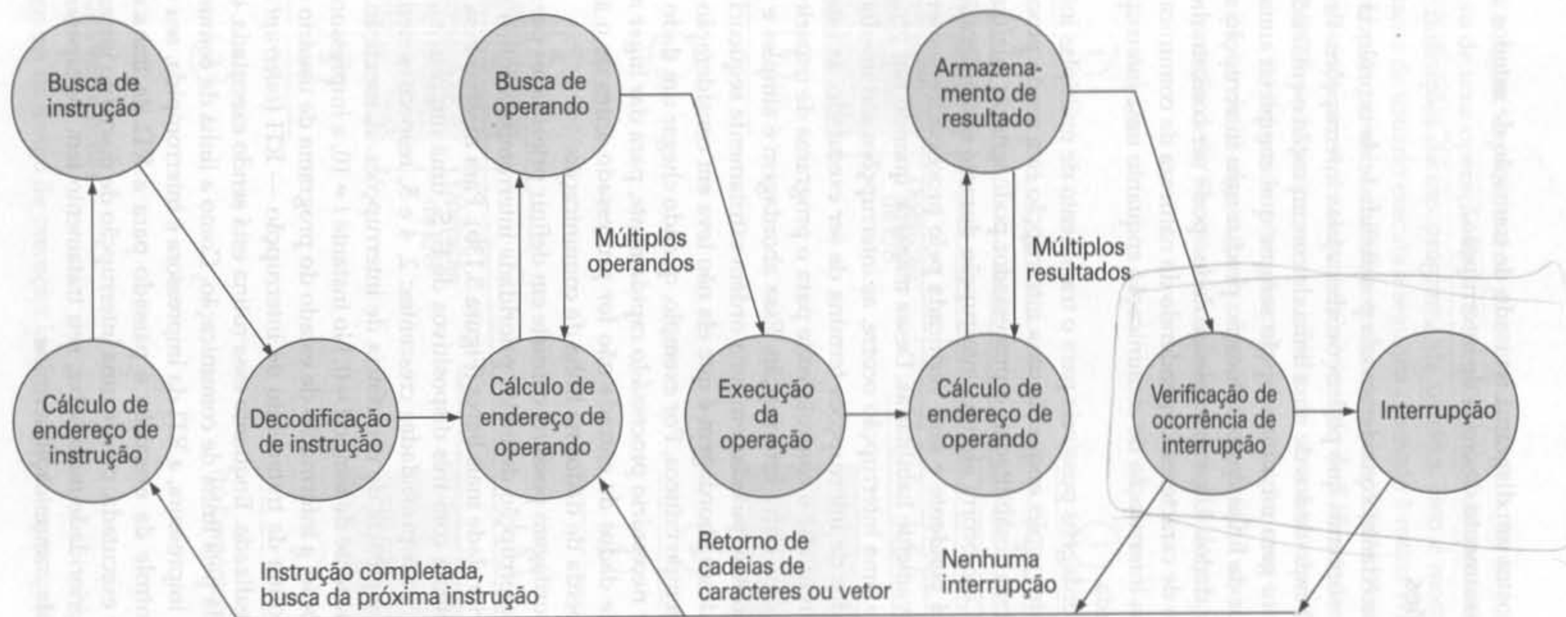
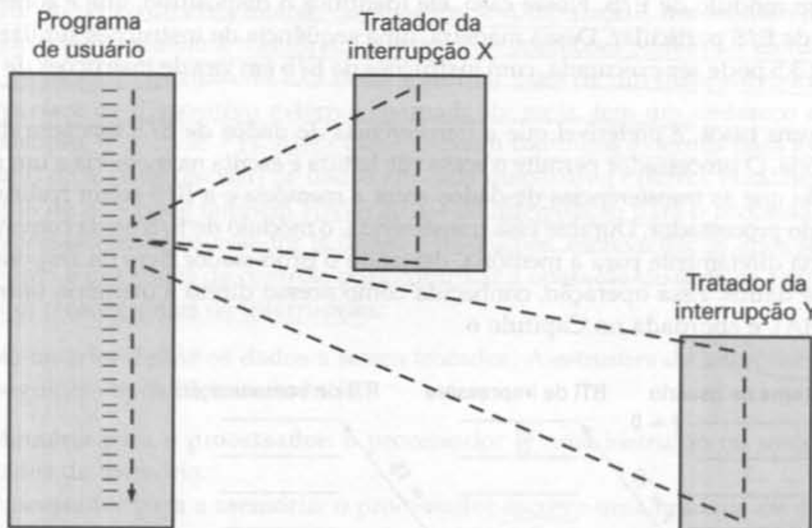
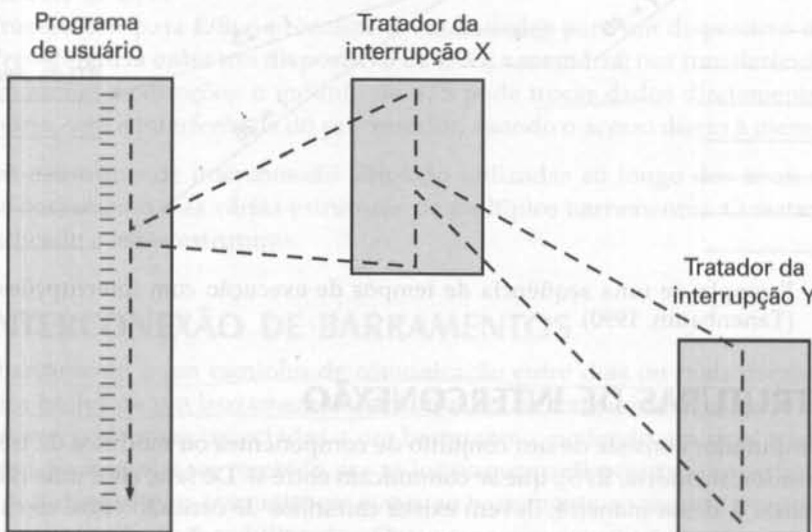


Figura 3.12 Diagrama de transição de estados de um ciclo de instrução com interrupções.

Quando isso ocorre ($t = 25$), o estado anterior do processador, correspondente à RTI da impressora, é restaurado. Entretanto, antes que qualquer instrução daquela rotina seja executada, o processador inicia o processamento da interrupção pendente e o controle do processador é transferido para a RTI do disco, de maior prioridade. Apenas quando a execução dessa rotina termina (no instante $t = 35$), a RTI da impressora reassume o controle. Finalmente, quando a execução dessa rotina termina (no instante $t = 40$), o controle retorna para o programa de usuário.



(a) Processamento seqüencial de interrupções



(b) Processamento aninhado de interrupções

Figura 3.13 Transferência de controle com interrupções múltiplas.

Funcionamento da E/S

Até agora discutimos o controle da operação do computador pelo processador, enfocando, principalmente, as interações entre o processador e a memória. A seguir, discutiremos brevemente o papel dos módulos de E/S, que será abordado mais detalhadamente no Capítulo 6.

Um módulo de E/S (por exemplo, um controlador de disco) pode trocar dados diretamente com o processador. Assim como o processador pode iniciar uma operação de leitura ou escrita na memória, designando um endereço específico, ele pode também ler ou escrever dados em um módulo de E/S. Nesse caso, ele identifica o dispositivo, que é controlado por um módulo de E/S particular. Dessa maneira, uma seqüência de instruções similar à mostrada na Figura 3.5 pode ser executada, com instruções de E/S em vez de instruções de referência à memória.

Em alguns casos, é preferível que a transferência de dados de E/S seja feita diretamente para a memória. O processador permite o acesso de leitura e escrita na memória a um módulo de E/S, de modo que as transferências de dados entre a memória e a E/S sejam realizadas sem a intervenção do processador. Durante essa transferência, o módulo de E/S envia comandos de leitura ou escrita diretamente para a memória, deixando o processador livre da responsabilidade pela troca de dados. Essa operação, conhecida como acesso direto à memória (*direct memory access* — DMA), é abordada no Capítulo 6.

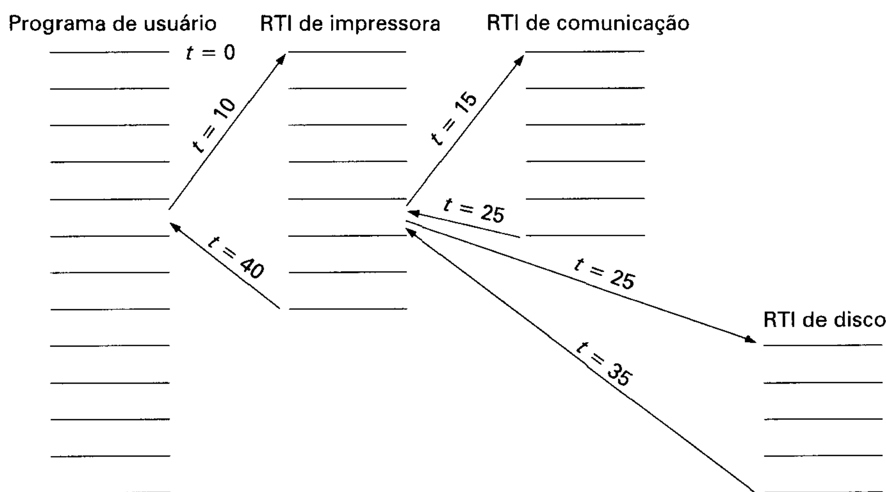


Figura 3.14 Exemplo de uma seqüência de tempos de execução com interrupções múltiplas (Tanenbaum, 1990).

3.3 ESTRUTURAS DE INTERCONEXÃO

Um computador consiste de um conjunto de componentes ou módulos de três tipos básicos (processador, memória, E/S), que se comunicam entre si. De fato, ele é uma rede de componentes básicos e, dessa maneira, devem existir caminhos de conexão entre esses módulos.

A coleção de caminhos que conectam os vários módulos é chamada *estrutura de interconexão*. O projeto dessa estrutura depende das informações trocadas entre os vários módulos.

A Figura 3.15 sugere os tipos de troca de informações necessárias, indicando as principais formas de entrada e saída para cada tipo de módulo:

- **Memória:** tipicamente, uma memória é composta de *N* palavras de um mesmo tamanho. Cada palavra possui um único endereço numérico (0, 1, ..., *N* – 1). Uma palavra pode ser lida ou escrita na memória. A natureza da operação é indicada por meio de sinais de controle. A posição de memória em que deve ser efetuada a operação é especificada por um endereço.
- **E/S:** do ponto de vista interno (ao sistema de computação), um módulo de E/S é funcionalmente similar à memória. Dois tipos de operações podem ser efetuados: leitura e escrita. Um módulo de E/S pode controlar mais de um dispositivo externo. Cada interface de dispositivo externo, chamada de *porta*, tem um endereço distinto (por exemplo, 0, 1, ..., *M* – 1). Além disso, existem caminhos externos para a entrada (leitura) ou a saída (escrita) de dados em cada dispositivo externo. Finalmente, um módulo de E/S pode também enviar sinais de interrupção para o processador.
- **Processador:** o processador lê dados e instruções, escreve dados após seu processamento e usa sinais de controle para controlar a operação do sistema todo. Pode também receber sinais de interrupção.

A lista anterior define os dados a serem trocados. A estrutura de interconexão deve suportar os seguintes tipos de transferência:

- **Memória para o processador:** o processador lê uma instrução ou uma unidade de dados da memória.
- **Processador para a memória:** o processador escreve uma unidade de dados na memória.
- **E/S para o processador:** o processador lê dados de um dispositivo de E/S via um módulo de E/S.
- **Processador para E/S:** o processador envia dados para um dispositivo de E/S.
- **Transferência entre um dispositivo de E/S e a memória:** nas transferências de dados em ambas as direções, o módulo de E/S pode trocar dados diretamente com a memória, sem a interferência do processador, usando o acesso direto à memória (DMA).

Várias estruturas de interconexão têm sido utilizadas ao longo dos anos. As mais comuns são o barramento e as várias estruturas de múltiplos barramentos. O restante deste capítulo é dedicado a essas estruturas.

3.4 INTERCONEXÃO DE BARRAMENTOS

Um barramento é um caminho de comunicação entre dois ou mais dispositivos. Uma característica básica de um barramento é ser um meio de transmissão compartilhado. Diversos dispositivos podem ser conectados a um barramento, podendo um sinal transmitido por qualquer dos dispositivos ser recebido por todos os outros dispositivos conectados ao barramento. Se dois dispositivos transmitirem sinais ao barramento ao mesmo tempo, esses sinais irão se sobrepor e serão então adulterados. Dessa maneira, para que a transmissão ocorra com sucesso, apenas um dispositivo pode transmitir sinais pelo barramento a cada instante.

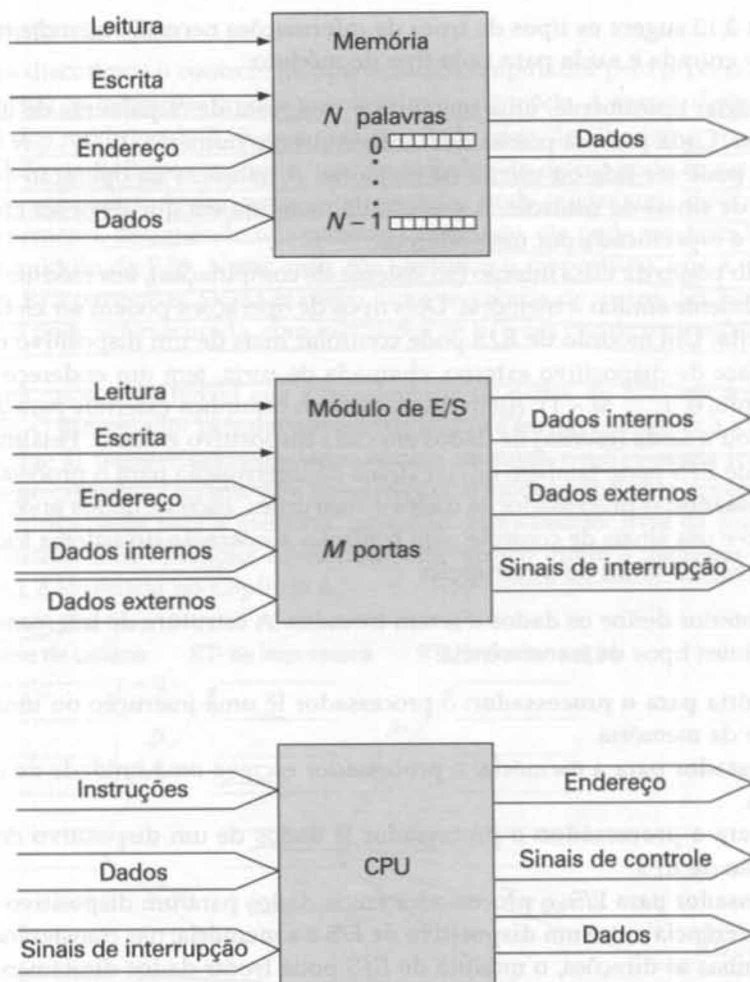


Figura 3.15 Módulos de um computador.

Tipicamente, um barramento consiste em vários caminhos ou linhas de comunicação, cada qual capaz de transmitir sinais que representam um único dígito binário, *0* ou *1*. Ao longo do tempo, uma seqüência de dígitos binários pode ser transmitida por meio de uma linha. As diversas linhas do barramento podem ser usadas, em conjunto, para transmitir vários dígitos binários simultaneamente (em paralelo). Por exemplo, uma unidade de dados de 8 bits pode ser transmitida por oito linhas do barramento.

Um sistema de computação contém diversos barramentos, que fornecem caminhos de comunicação entre os seus componentes, nos vários níveis da hierarquia do sistema. O barramento usado para conectar os componentes principais do computador (processador, memória, E/S) é conhecido como *barramento do sistema*. As estruturas de interconexão mais comuns são baseadas no uso de um ou mais barramentos do sistema.

Estrutura de barramentos

Um barramento do sistema contém, tipicamente, de 50 a 100 linhas distintas. Cada linha possui uma função ou significado particular. Embora existam diferentes projetos de barramento, as linhas de um barramento podem ser classificadas em três grupos funcionais (Figura 3.16): linhas de dados, linhas de endereço e linhas de controle. Além disso, devem existir linhas para a distribuição de energia para os módulos conectados no barramento.

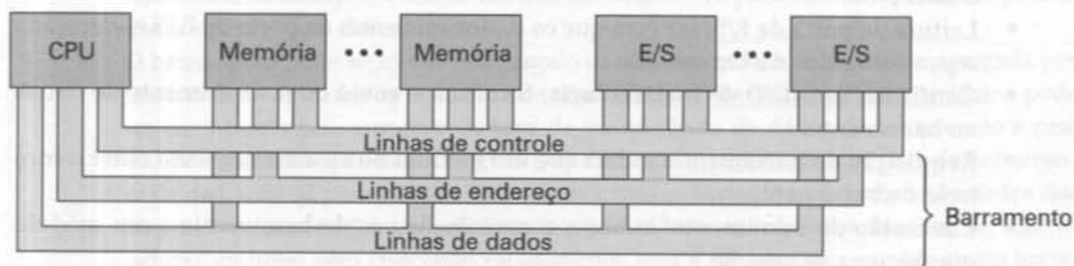


Figura 3.16 Esquema de interconexão de barramento.

As *linhas de dados* fornecem um caminho para a transferência de dados entre os módulos do sistema. Esse conjunto de linhas é denominado *barramento de dados*. O barramento de dados contém tipicamente 8, 16 ou 32 linhas; o número de linhas é conhecido como a *largura* do barramento de dados. Como cada linha pode conduzir apenas 1 bit por vez, o número de linhas determina quantos bits podem ser transferidos de uma vez. A largura do barramento de dados constitui um parâmetro fundamental para o desempenho global do sistema. Por exemplo, se o barramento de dados tem largura de 8 bits e cada instrução tem tamanho de 16 bits, o processador tem de acessar duas vezes o módulo de memória em cada ciclo de instrução.

As *linhas de endereço* são utilizadas para designar a fonte ou o destino dos dados transferidos pelo *barramento de dados*. Por exemplo, quando o processador deseja ler uma palavra (de 8, 16 ou 32 bits) da memória, ele coloca o endereço da palavra desejada nas linhas de endereço. A largura do barramento de endereço determina a capacidade máxima da memória do sistema. Em geral, as linhas de endereço também são empregadas para endereçar as portas de E/S. Tipicamente, os bits mais significativos são utilizados para identificar um módulo particular do sistema conectado ao barramento e os bits menos significativos identificam uma posição na memória ou uma porta de E/S nesse módulo. Por exemplo, em um barramento de 8 bits, os endereços menores ou iguais a 01111111 podem ser endereços de posições de uma memória (módulo 0) com 128 palavras, enquanto os endereços maiores ou iguais a 10000000 podem ser relativos a endereços de dispositivos em um módulo de E/S (módulo 1).

As *linhas de controle* são usadas para controlar o acesso e a utilização das linhas de dados e de endereço. Como as linhas de dados e de endereço são compartilhadas por todos os componentes, deve existir uma maneira de controlar sua utilização. Os sinais de controle são utilizados tanto para transmitir comandos quanto para transmitir informações de temporização entre os módulos do sistema. Os sinais de temporização indicam a validade das informações de dados e de endereço. Os sinais de comando especificam as operações a serem executadas.

Linhas de controle típicas incluem as seguintes:

- **Escrita na memória:** faz com que os dados existentes nas linhas de dados do barramento sejam gravados na posição de memória especificada nas linhas de endereço.
- **Leitura de memória:** faz com que o valor armazenado no endereço da memória especificada nas linhas de endereço seja colocado nas linhas de dados do barramento.
- **Escrita em porta de E/S:** causa o envio dos dados do barramento para a porta de E/S endereçada.
- **Leitura de porta de E/S:** faz com que os dados existentes na porta de E/S endereçada sejam colocados no barramento.
- **Confirmação (*ACK*) de transferência:** confirma o envio ou o recebimento de dados no barramento.
- **Requisição do barramento:** indica que um módulo do sistema necessita obter o controle do barramento.
- **Concessão do barramento:** indica a concessão de uso do barramento a um módulo que fez uma requisição.
- **Requisição de interrupção:** indica a existência de uma interrupção pendente.
- **Confirmação (*ACK*) de interrupção:** confirma o reconhecimento de uma interrupção pendente.
- **Relógio:** utilizado para temporização de operações.
- **Inicialização (*reset*):** inicializa todos os módulos do sistema.

O barramento opera do modo especificado a seguir. Quando um módulo do sistema deseja enviar dados para outro, ele deve: (1) obter o controle do barramento e (2) transferir os dados por meio do barramento. Quando um módulo deseja requisitar dados de outro módulo, ele deve: (1) obter o controle do barramento e (2) transferir uma requisição para o outro módulo por meio das linhas de endereço e de controle apropriadas. Em seguida, deve esperar que o outro módulo envie os dados requisitados.

Fisicamente, o barramento do sistema é na verdade um conjunto de condutores elétricos paralelos. Esses condutores são linhas de metal impressas em um cartão ou placa (placa de circuito impresso). O barramento se estende por todos os componentes do sistema, cada um dos quais se liga a algumas ou a todas as linhas do barramento. Um arranjo físico muito comum é mostrado na Figura 3.17. Nesse exemplo, o barramento consiste em duas colunas verticais de condutores. Em intervalos regulares ao longo das colunas, existem pontos de fixação, na forma de ranhuras, que se estendem horizontalmente para apoiar as placas de circuito impresso. Cada um dos principais componentes do sistema ocupa uma ou mais placas e se conecta ao barramento por meio dessas ranhuras. O arranjo todo é alojado em um chassi.

Esse arranjo é bastante conveniente, pois permite que uma configuração pequena seja posteriormente expandida (incluindo mais memória ou mais E/S), pela adição de novas placas. Caso ocorra uma falha em um componente de alguma placa, essa placa pode ser facilmente removida e substituída.

Hierarquia de múltiplos barramentos

O desempenho do sistema pode ser prejudicado caso o número de dispositivos conectados a um barramento seja muito grande. As principais causas são:

1. Em geral, quanto maior o número de dispositivos conectados, maior é o comprimento de um barramento e, assim, maior o atraso na propagação de sinais. Esse atraso determina o tempo gasto para que um dispositivo obtenha o controle do barramento. Quando o controle do barramento passa muitas vezes de um dispositivo para outro, esses atrasos podem afetar seriamente o desempenho.
2. O barramento pode se tornar um gargalo do sistema quando a demanda agregada por transferência de dados se aproxima da capacidade do barramento. Esse problema pode ser contornado pelo aumento da taxa de transferência de dados do barramento e pelo uso de barramentos de maior largura (por exemplo, aumentando a largura do barramento de dados de 32 para 64 bits). Entretanto, como as taxas de transferência de dados dos dispositivos conectados (ou seja, interfaces de rede e controladores de vídeo ou controladores gráficos) vêm crescendo rapidamente, essa é uma batalha que um único barramento certamente está fadado a perder.

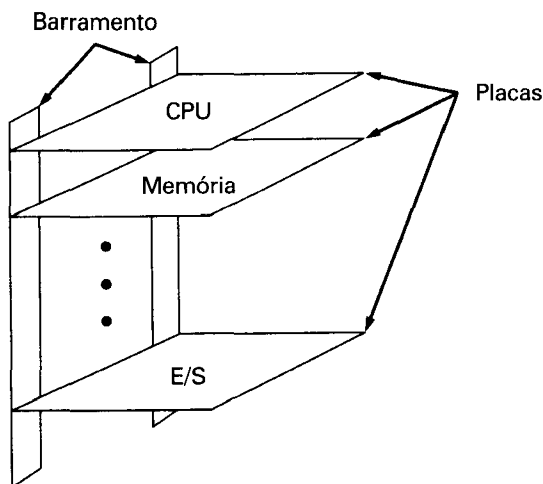


Figura 3.17 Estrutura física típica de uma arquitetura de barramentos.

Desse modo, a maioria dos sistemas de computação utiliza múltiplos barramentos, geralmente dispostos de maneira hierárquica. A Figura 3.18a mostra uma estrutura tradicional típica. Um barramento local conecta o processador a uma memória cache e pode conter um ou mais dispositivos locais. O controlador da memória cache conecta a memória cache não apenas a esse barramento local, mas também a um barramento do sistema, ao qual são conectados os módulos da memória principal. Como veremos no Capítulo 4, o uso de uma estrutura de memória cache evita que o processador tenha de acessar freqüentemente a memória principal. Portanto, a memória principal pode ser conectada apenas ao barramento do sistema, não precisando ser conectada ao barramento local. Dessa maneira, as transferências de dados entre os componentes de E/S e a memória principal por meio do barramento do sistema não interferem na atividade do processador.

Os controladores de E/S podem ser conectados diretamente ao barramento do sistema. Entretanto, uma solução mais eficiente é utilizar um ou mais barramentos de expansão. Uma interface de expansão de barramentos serve como área de armazenamento temporário dos dados transferidos entre o barramento do sistema e os controladores de E/S conectados ao barramento de expansão. Esse arranjo permite ao sistema conectar uma grande variedade de dispositivos de E/S e, ao mesmo tempo, isolar o tráfego entre o processador e a memória do tráfego de E/S.

A Figura 3.18a mostra alguns exemplos típicos de dispositivos de E/S que podem ser conectados ao barramento de expansão. As conexões de rede incluem redes locais (*local area networks* — LANs), tais como uma rede Ethernet de 10 Mbps, assim como redes geograficamente distribuídas (*wide area networks* — WANs), tais como as redes de comutação de pacotes. A interface SCSI (*small computer system interface* — interface de sistema de computadores pequenos) é um tipo de barramento utilizado para a conexão de discos locais e de outros periféricos. Uma porta serial pode ser utilizada para a conexão de uma impressora ou de um scanner.

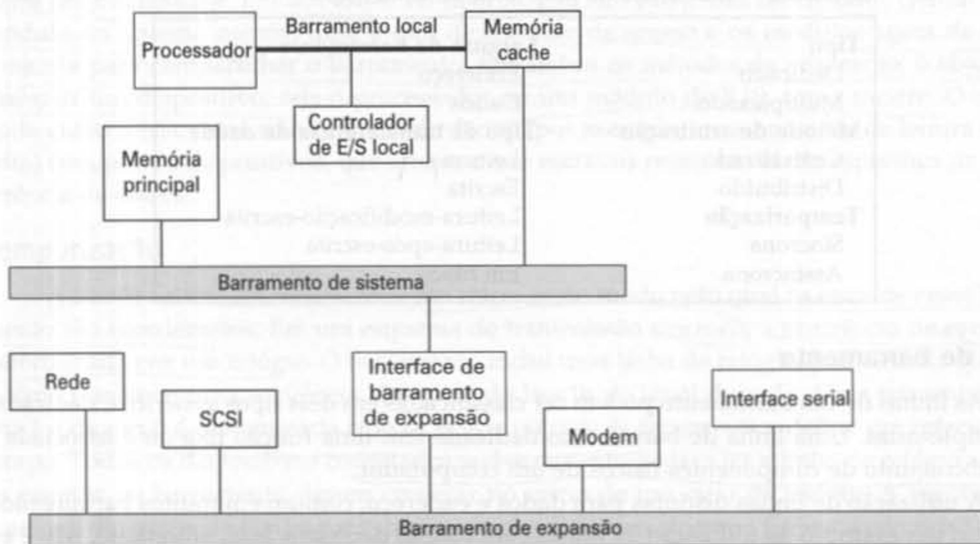
Embora essa arquitetura tradicional de barramentos seja razoavelmente eficiente, ela não é satisfatória para a conexão de dispositivos de E/S mais modernos, que apresentam desempenho cada vez maior. A abordagem mais adotada pela indústria para satisfazer as demandas crescentes por melhor desempenho consiste em utilizar um barramento de alta velocidade que seja estreitamente integrado ao resto do sistema, requerendo apenas uma ponte entre o barramento do processador e o barramento de alta velocidade. Esse arranjo é conhecido como arquitetura de mezanino.

A Figura 3.18b mostra uma implementação típica dessa abordagem. Um barramento local conecta o processador ao controlador da memória cache, o qual, por sua vez, é conectado a um barramento do sistema que contém a memória principal. O controlador da memória cache é integrado a uma ponte, ou dispositivo de armazenamento temporário, que se conecta ao barramento de alta velocidade. Esse barramento permite a conexão de redes locais de alta velocidade, como uma rede Fast Ethernet de 100 Mbps, controladores de vídeo de estações de trabalho gráficas, assim como controladores de interface para barramentos periféricos locais, incluindo SCSI e *FireWire*. O *FireWire* é um barramento de alta velocidade especificamente projetado para dispositivos de E/S de alta capacidade. Os dispositivos de menor velocidade ainda são conectados a um barramento de expansão, comunicando-se com o barramento de alta velocidade por meio de uma interface que fornece armazenamento temporário de dados entre esses dois tipos de barramento.

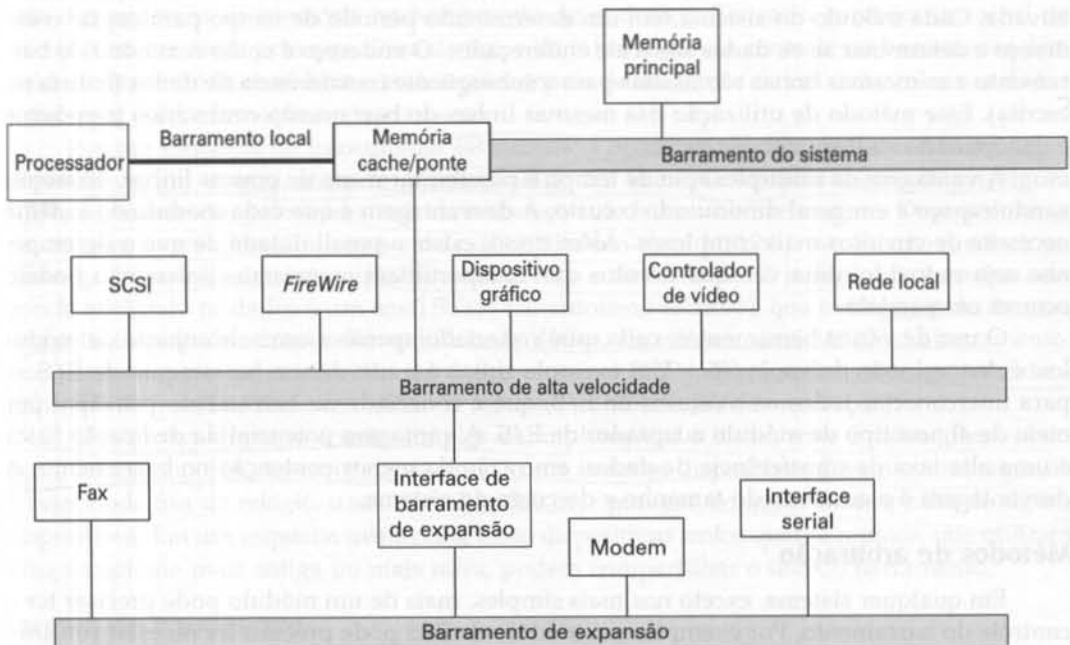
A vantagem dessa configuração é que o barramento de alta velocidade permite maior integração entre o processador e os dispositivos com alta demanda de tráfego e ao mesmo tempo é independente do processador. Assim, diferenças de velocidade e de definição de sinais entre o processador e o barramento de alta velocidade podem ser toleradas. Mudanças na arquitetura do processador não afetam o barramento de alta velocidade e vice-versa.

Elementos de projeto de barramentos

Embora exista uma variedade de diferentes implementações de barramentos, poucos parâmetros ou elementos de projeto básicos podem ser empregados para classificar e diferenciar barramentos. A Tabela 3.2 enumera esses parâmetros básicos.



(a) Arquitetura de barramento tradicional



(b) Arquitetura de alto desempenho

Figura 3.18 Exemplos de configurações de barramento.

Tabela 3.2 Elementos de projeto de barramentos

Tipo	**Largura do barramento**
Dedicado	Endereço
Multiplexado	Dados
Método de arbitração	**Tipo de transferência de dados**
Centralizado	Leitura
Distribuído	Escrita
Temporização	Leitura-modificação-escrita
Síncrona	Leitura-após-escrita
Assíncrona	Em bloco

Tipos de barramento

As linhas de um barramento podem ser classificadas em dois tipos genéricos: dedicadas e multiplexadas. Uma linha de barramento dedicada tem uma função fixa ou é associada a um subconjunto de componentes físicos de um computador.

A utilização de linhas distintas para dados e endereço, comum em muitos barramentos, constitui um exemplo de utilização de linhas com função dedicada. Isso, entretanto, não é essencial: informações de dados e de endereço podem ser transmitidas por meio do mesmo conjunto de linhas, utilizando-se uma linha de controle de *Endereço Válido*. No início de uma transferência de dados, o endereço é colocado no barramento e a linha de endereço válido é ativada. Cada módulo do sistema tem um determinado período de tempo para copiar o endereço e determinar se os dados são a ele endereçados. O endereço é então removido do barramento e as mesmas linhas são usadas para a subseqüente transferência de dados (leitura ou escrita). Esse método de utilização das mesmas linhas do barramento com vários propósitos é denominado *multiplexação de tempo*.

A vantagem da multiplexação de tempo é possibilitar o uso de poucas linhas, economizando espaço e em geral diminuindo o custo. A desvantagem é que cada módulo do sistema necessita de circuitos mais complexos. Além disso, existe a possibilidade de que o desempenho seja reduzido, uma vez que eventos que compartilham as mesmas linhas não podem ocorrer em paralelo.

O uso de vários barramentos, cada qual conectado apenas a um subconjunto de módulos, é denominado *dedicação física*. Um exemplo típico é o uso de um barramento de E/S — para interconectar todos os módulos de E/S, que é conectado ao barramento principal por meio de algum tipo de módulo adaptador de E/S. A vantagem potencial da dedicação física é uma alta taxa de transferência de dados, em razão da menor contenção no barramento. A desvantagem é o aumento do tamanho e do custo do sistema.

Métodos de arbitração

Em qualquer sistema, exceto nos mais simples, mais de um módulo pode precisar ter o controle do barramento. Por exemplo, um módulo de E/S pode precisar ler ou escrever diretamente na memória, sem enviar dados ao processador. Como apenas uma unidade pode realizar uma transmissão por meio do barramento de cada vez, é necessário utilizar algum método de arbitração. Os vários métodos podem ser classificados como centralizados ou distribuídos. Em um esquema centralizado, um único dispositivo de hardware, conhecido como *controlador de barramento* ou *árbitro*, é responsável por alocar tempo de utilização do barramento a cada módulo do sistema. Esse dispositivo pode constituir um módulo separado ou fazer

parte do processador. Em um esquema distribuído, não existe um controlador central. Cada módulo do sistema contém uma lógica de controle de acesso e os módulos agem de forma conjunta para compartilhar o barramento. Em ambos os métodos de arbitração, o objetivo é designar um dispositivo, seja o processador ou um módulo de E/S, como mestre. O mestre pode então iniciar uma transferência de dados (por exemplo, uma operação de leitura ou escrita) com outros dispositivos, que atuam como escravos nessa atividade específica de transferência de dados.

Temporização

A temporização de um barramento refere-se ao modo pelo qual os eventos nesse barramento são coordenados. Em um esquema de transmissão síncrona, a ocorrência de eventos é determinada por um relógio. O barramento inclui uma linha de relógio, por meio da qual um relógio transmite uma seqüência alternada de 1s e 0s de igual duração. Uma transmissão de um *1* e de um *0* é denominada *ciclo de relógio* ou *ciclo de barramento* e define um intervalo de tempo. Todos os dispositivos conectados ao barramento podem ler a linha de relógio e todos os eventos no barramento devem começar no início de um ciclo de relógio. A Figura 3.19a mostra o diagrama de tempo para uma operação de leitura síncrona (veja o Apêndice 3A para uma descrição dos diagramas de tempo). Outros sinais do barramento podem ser emitidos no início do ciclo de relógio (com um leve atraso de reação). A maioria dos eventos dura um único ciclo de relógio. Nesse exemplo simples, o processador emite um sinal de leitura e coloca um endereço de memória no barramento de endereço. Ele emite também um sinal de iniciar para marcar a presença do endereço e de informação de controle no barramento. Um módulo de memória reconhece o endereço e, depois de um atraso de um ciclo, coloca os dados e um sinal de confirmação no barramento.

Em um esquema de transmissão assíncrona, a ocorrência de um evento no barramento depende de um evento ocorrido anteriormente. No exemplo simples mostrado na Figura 3.19b, o processador coloca os sinais de endereço e de leitura no barramento. Depois de uma pausa para a estabilização desses sinais, ele emite um sinal MSYN (sincronismo mestre), indicando a presença de sinais válidos de endereço e de controle. O módulo de memória responde enviando os dados e um sinal SSYN (sincronismo escravo), que indica o envio de uma resposta. Depois de o mestre ler as linhas de dados, ele retira o sinal MSYN do barramento. Isso faz com que a memória retire os dados e o sinal SSYN. Finalmente, uma vez retirada a linha SSYN, o mestre remove o sinal de leitura e a informação de endereço.

O esquema síncrono é mais simples de implementar e testar, mas é menos flexível do que o assíncrono. Em um esquema síncrono, como todos os dispositivos devem operar segundo a velocidade fixa do relógio, o sistema não pode tirar proveito do maior desempenho de alguns dispositivos. Em um esquema assíncrono, tanto dispositivos lentos quanto rápidos, que utilizam uma tecnologia mais antiga ou mais nova, podem compartilhar o uso do barramento.

Largura do barramento

Já discutimos o conceito de largura do barramento. A largura do barramento de dados tem impacto sobre o desempenho do sistema: quanto maior a largura do barramento de dados, maior o número de bits transferidos de cada vez. A largura do barramento de endereço tem impacto sobre a capacidade do sistema: quanto maior a largura do barramento de endereço, maior é o número de posições de memória que podem ser endereçadas.

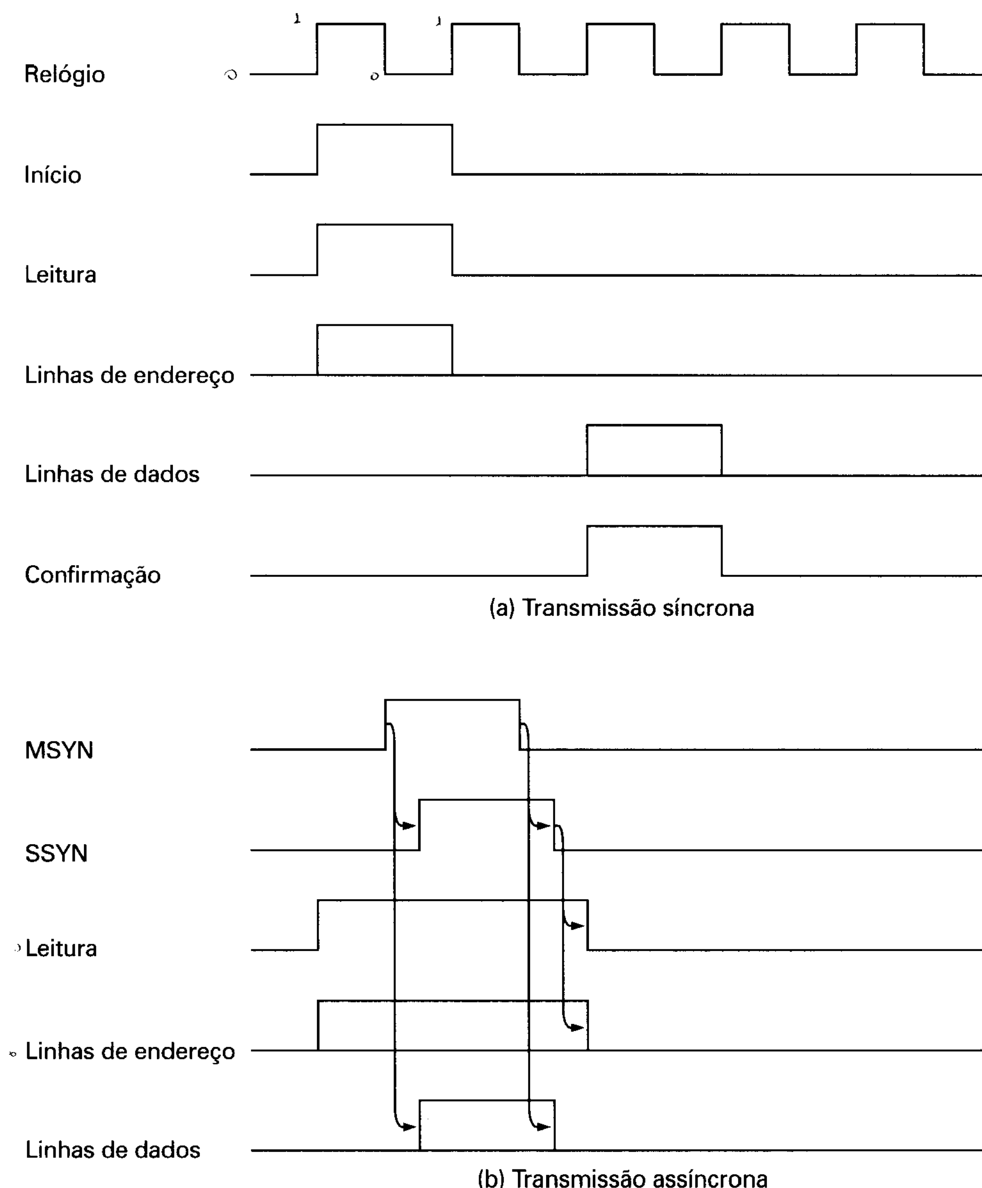


Figura 3.19 Temporização de uma operação de leitura.

Tipos de transferências de dados

Como mostra a Figura 3.20, um barramento permite vários tipos de transferência de dados. Qualquer barramento admite transferências de dados para escrita (mestre para escravo) ou leitura (escravo para mestre). No caso de um barramento de dados/endereço multiplexado, o barramento é usado primeiramente para a especificação do endereço e depois para a transferência de dados. Em uma operação de leitura, normalmente existe um tempo de espera, enquanto um valor está sendo buscado pelo escravo para ser colocado no barramento. Tanto

na leitura quanto na escrita poderá também haver um atraso caso seja necessária uma arbitração para obter o controle do barramento a fim de efetuar uma operação (isto é, para obter o controle do barramento a fim de enviar uma requisição de leitura ou escrita e depois obter o controle novamente para efetuar a leitura ou a escrita).

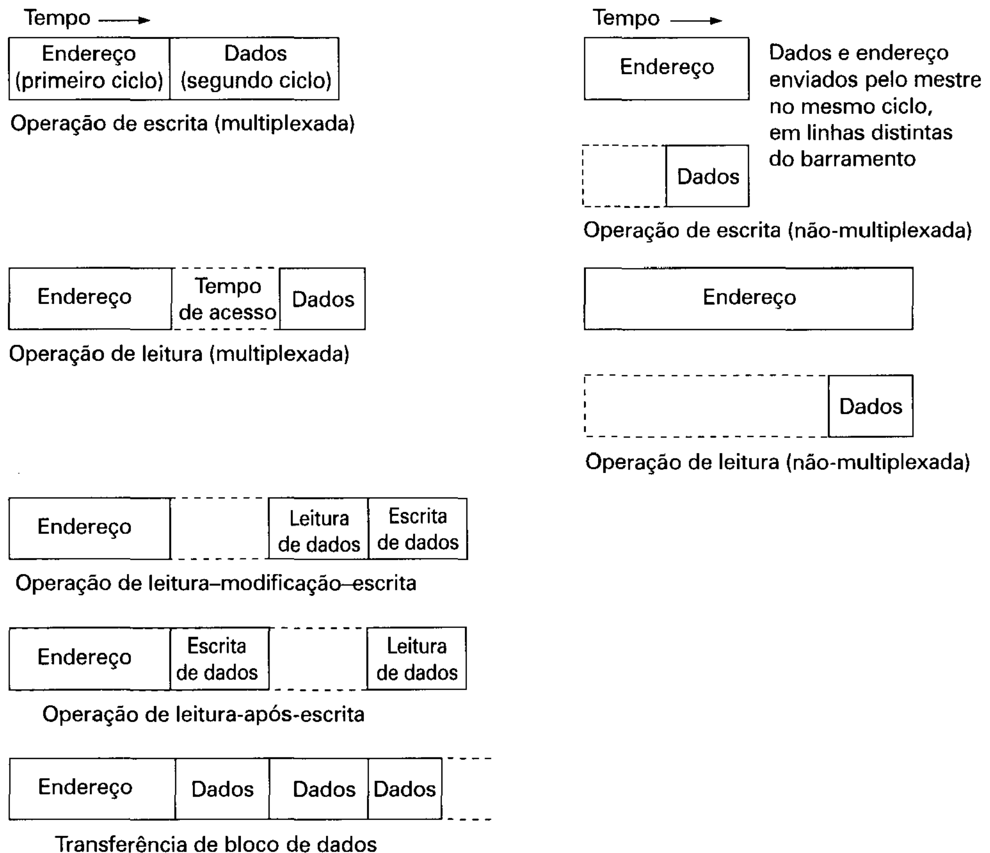


Figura 3.20 Tipos de transferências de dados no barramento (Goor, 1989).

No caso de linhas de dados e de endereço dedicadas, o endereço é colocado no barramento de endereço e permanece lá até que os dados sejam colocados no barramento de dados. Em uma operação de escrita, o mestre coloca os dados no barramento de dados assim que as linhas de endereço tenham se estabilizado e o escravo tenha reconhecido o endereço enviado. Em uma operação de leitura, o escravo coloca os dados no barramento de dados assim que reconheça o endereço e busque os dados requeridos.

Alguns barramentos permitem também diversas combinações dessas operações. Uma operação de leitura-modificação-escrita é simplesmente uma leitura seguida imediatamente por uma escrita sobre o mesmo endereço. O endereço é transmitido apenas uma vez, no início da operação. A operação completa é normalmente indivisível, para evitar o acesso ao elemento de dados por outros mestres potenciais do barramento. O principal uso dessa capacidade é proteger recursos de memória compartilhada em um sistema de multiprogramação (veja o Capítulo 7).

Uma operação de leitura-após-escrita é uma operação indivisível que consiste em uma escrita seguida imediatamente por uma leitura no mesmo endereço. A operação de leitura pode ser feita com o propósito de verificação.

Alguns sistemas de barramento também oferecem transferências de blocos de dados. Nesse caso, um ciclo de endereço é seguido por *n* ciclos de dados. O primeiro item de dados é transferido de ou para o endereço especificado; os restantes são transferidos de ou para endereços subseqüentes.

3.5 PCI

O barramento PCI (*peripheral component interconnect* — interconexão de componentes periféricos) é um barramento de grande largura de banda, independente do processador utilizado, que pode funcionar como um barramento periférico ou mezanino. Comparando as especificações com outros barramentos mais comuns, o PCI apresenta melhor desempenho do sistema para subsistemas de E/S de alta velocidade (tais como adaptadores de videográficos, controladores de interface de rede e controladores de disco). O padrão atual permite o uso de até 64 linhas de dados de 66 MHz, com uma taxa bruta de transferência de dados de 528 Mbytes/s ou 4,224 Gbps. Mas não é apenas a alta velocidade que torna o PCI atraente. O PCI é especificamente projetado para satisfazer os requisitos de custo de E/S dos sistemas modernos; sua implementação requer poucas pastilhas (chips), e outros barramentos podem ser conectados ao barramento PCI.

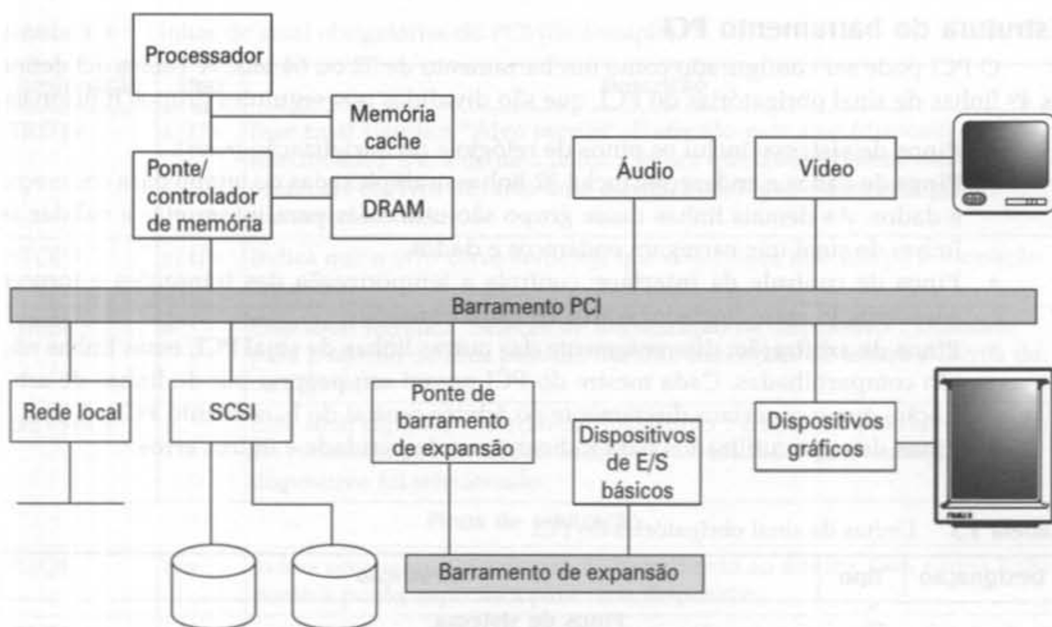
A Intel iniciou o projeto do PCI em 1990, visando utilizá-lo em seus sistemas baseados no processador Pentium. Logo depois, ela liberou todas as patentes para domínio público e promoveu a criação de uma associação de fabricantes, a PCI SIG, para dar continuidade ao desenvolvimento e manter a compatibilidade das especificações PCI. Com isso, o PCI vem sendo cada vez mais utilizado em computadores pessoais, estações de trabalho e servidores. A versão PCI 2.1 foi lançada em 1995*. Como a especificação é de domínio público e grande parte da indústria de microprocessadores e de periféricos adota essa especificação, os diversos produtos PCI construídos por diferentes fabricantes são compatíveis entre si.

O PCI é projetado para trabalhar com uma variedade de configurações de microprocessadores, incluindo sistemas com um único ou com múltiplos processadores. Dessa maneira, ele fornece um conjunto de funções de propósito geral. O PCI utiliza o esquema síncrono de transferência de dados e um esquema de arbitração centralizado.

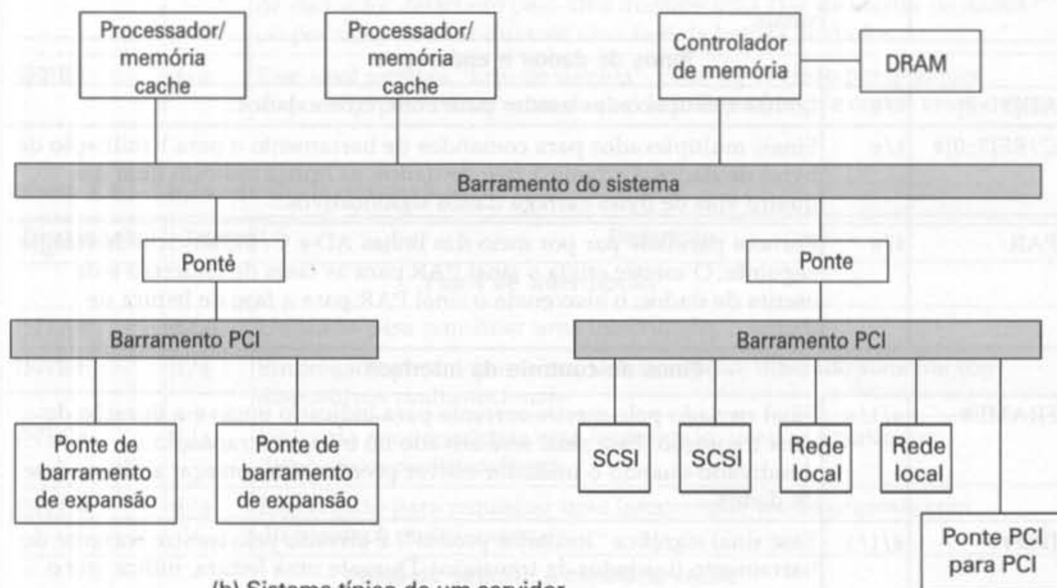
A Figura 3.21a mostra uma utilização típica do PCI em um sistema com um único processador. Um controlador DRAM combinado com uma ponte para o barramento PCI oferece um acoplamento adequado com o processador, possibilitando a transferência de dados à alta velocidade. A ponte atua como uma área de armazenamento temporário de dados, permitindo que a velocidade do barramento PCI seja diferente da capacidade de transferência de E/S do processador. Em um sistema com múltiplos processadores (Figura 3.21b), uma ou mais configurações de PCI podem ser conectadas ao barramento do sistema, por meio de pontes. O barramento do sistema dá suporte apenas para as unidades de processador/memória cache, memória principal e pontes PCI. Mais uma vez, o uso de pontes mantém o PCI independente da velocidade do processador e possibilita receber e enviar dados rapidamente.

---

* N.R.T.: Em 25/1/1999 foi lançada uma nova versão do PCI, a versão 2.2. Uma extensão do PCI chamada PCI-X, com barramento de 64 bits e taxa de transferência de dados de até 1,066 GB/s, foi definida em 27/9/1999. Diversos produtos compatíveis com o PCI-X já foram lançados durante o ano de 2001 pela Compaq, FuturePlus Systems e Catalyst, por exemplo.



(a) Sistema típico de um computador de mesa



(b) Sistema típico de um servidor

Figura 3.21 Exemplos de configurações com barramento PCI.

Estrutura do barramento PCI

O PCI pode ser configurado como um barramento de 32 ou 64 bits. A Tabela 3.3 define as 49 linhas de sinal obrigatórias do PCI, que são divididas nos seguintes grupos funcionais:

- **Pinos de sistema:** inclui os pinos de relógio e de inicialização (*reset*).
- **Pinos de dados e endereços:** inclui 32 linhas multiplexadas no tempo para endereços e dados. As demais linhas desse grupo são utilizadas para interpretar e validar as linhas de sinal que carregam endereços e dados.
- **Pinos de controle da interface:** controla a temporização das transações e fornece coordenação entre iniciador e alvo de cada transação.
- **Pinos de arbitração:** diferentemente das outras linhas de sinal PCI, essas linhas não são compartilhadas. Cada mestre do PCI possui seu próprio par de linhas de arbitração, que o conectam diretamente ao árbitro central do barramento PCI.
- **Pinos de erros:** utilizados para indicar erros de paridade e outros erros.

Tabela 3.3 Linhas de sinal obrigatórias do PCI

Designação	Tipo	Descrição
		Pinos de sistema
CLK	e	Fornece a temporização para todas as transações e todas as entradas são amostradas na ativação do sinal. Admite a operação com taxas de relógio de até 33 MHz.
RST#	e	Inicializa todos os registradores específicos do PCI, seqüenciadores e sinais.
		Pinos de dados e endereços
AD[31::0]	t/e	Linhas multiplexadas usadas para endereços e dados.
C/BE[3::0]#	t/e	Sinais multiplexados para comandos de barramento e para habilitação de bytes de dados. Durante a fase de dados, as linhas indicam qual das quatro vias de bytes carrega dados significativos.
PAR	t/e	Fornece paridade par por meio das linhas AD e C/BE no ciclo de relógio seguinte. O mestre envia o sinal PAR para as fases de endereço e de escrita de dados; o alvo envia o sinal PAR para a fase de leitura de dados.
		Pinos de controle da interface
FRAME#	s/t/s	Sinal enviado pelo mestre corrente para indicar o início e a duração de uma transação. Esse sinal será ativado no início da transação e desativado quando o iniciador estiver pronto para começar a última fase de dados.
IRDY#	s/t/s	Esse sinal significa "Iniciador pronto". É ativado pelo mestre corrente do barramento (iniciador da transição). Durante uma leitura, indica que o mestre está preparado para receber dados; durante uma escrita, indica que dados válidos estão presentes em AD.

Tabela 3.3 Linhas de sinal obrigatórias do PCI (*continuação*)

Designação	Tipo	Descrição
TRDY#	s/t/s	Esse sinal significa "Alvo pronto". É ativado pelo alvo (dispositivo selecionado). Durante uma leitura, indica que dados válidos estão presentes em AD; durante uma escrita, indica que o alvo está pronto para receber dados.
STOP#	s/t/s	Indica que o alvo corrente deseja que o iniciador interrompa a transação em curso.
IDSEL	e	Esse sinal significa "Seleção de inicialização de dispositivo". Utilizado para a seleção de uma pastilha durante transações de leitura e escrita de configuração.
DEVSEL#	e	Esse sinal significa "Seleção de dispositivo". É enviado pelo alvo quando ele reconhece seu endereço. Indica para o iniciador atual se algum dispositivo foi selecionado.
Pinos de arbitração		
REQ#	t/e	Indica uma requisição de uso do barramento ao árbitro. Essa é uma linha ponto a ponto, específica para cada dispositivo.
GNT#	t/e	Indica ao dispositivo que o árbitro permitiu o acesso ao barramento. Essa é uma linha ponto a ponto, específica para cada dispositivo.
Pinos de erros		
PERR#	s/t/s	Esse sinal significa "Erro de paridade". Indica que um erro de paridade de dados foi detectado pelo alvo durante uma fase de escrita de dados ou por um iniciador durante uma fase de leitura de dados.
SERR#	d/a	Esse sinal significa "Erro de sistema". Pode ser enviado por qualquer dispositivo para relatar erros de paridade de endereço e outros erros críticos.

Tabela 3.4 Linhas de sinais opcionais do PCI

Designação	Tipo	Descrição
Pinos de interrupção		
INTA#	d/a	Utilizado para requisitar uma interrupção.
INTB#	d/a	Empregado para requisitar uma interrupção; utilizado somente com dispositivos multifuncionais.
INTC#	d/a	Utilizado para requisitar uma interrupção; usado somente com dispositivos multifuncionais.
INTD#	d/a	Empregado para requisitar uma interrupção; usado somente com dispositivos multifuncionais.
Pinos de suporte à memória cache		
SBO#	e/s	Indica que uma linha de cache modificada foi encontrada na memória cache.
SDONE	e/s	Indica o estado da pesquisa (*snoop*) pelo endereço corrente. É ativado quando a pesquisa é completada.

(*continua*)

Tabela 3.4 Linhas de sinais opcionais do PCI (*continuação*)

Designação	Tipo	Descrição
Pinos de extensão do barramento		
AD[63::32]	t/e	Linhas multiplexadas para endereço e dados, para estender o barramento até 64 bits.
C/BE[7::4]#	t/e	Sinais multiplexados para comandos de barramento e para habilitação de bytes de dados. Durante a fase de endereço, as linhas fornecem comandos de barramento adicionais. Durante a fase de dados, as linhas indicam qual das quatro vias de bytes da extensão carregam dados significativos.
REQ64#	s/t/s	Usado para requisitar uma transferência de 64 bits.
ACK64#	s/t/s	Indica que o dispositivo-alvo deseja efetuar uma transferência de 64 bits.
PAR64	t/e	Apresenta paridade por meio dos pinos AD e C/BE da extensão no ciclo de relógio seguinte.
JTAG/pinos de teste		
TCK	e	Teste de relógio. Utilizado para informação de estado do relógio e para dados de entrada e saída de teste de dispositivo.
TDI	e	Entrada de teste. Empregado para transmissão serial de dados de teste e de instruções para o dispositivo.
TDO	s	Saída de teste. Utilizado para transmissão serial de dados de teste e de instruções do dispositivo.
TMS	e	Seleção de modo de teste. Utilizado para controlar o estado do controlador de portas de teste.
TRST#	e	Utilizado para inicializar o controlador de portas de teste.

e Apenas sinal de entrada
s Apenas sinal de saída
t/e Sinal de E/S, bidirecional, de três estados
s/t/s Sinal de três estados sustentado, enviado por um único proprietário de cada vez
d/a Dreno aberto: pode ser compartilhado por múltiplos dispositivos como um circuito *wired-OR*
O estado ativo do sinal ocorre em baixa voltagem

Além disso, a especificação do PCI define 51 linhas de sinal opcionais (Tabela 3.4), divididas nos seguintes grupos funcionais:

- **Pinos de interrupção:** esses pinos são disponíveis aos dispositivos PCI que precisam gerar interrupções. Assim como os pinos de arbitração, essas linhas não são compartilhadas; cada dispositivo tem sua própria linha de interrupção (ou linhas) para um controlador de interrupção.
- **Pinos de suporte à memória cache:** esses pinos são necessários para o funcionamento de uma memória no PCI que possa ser armazenada em uma memória cache, seja do processador ou de outro dispositivo. Eles permitem a implementação dos protocolos de cache *snoopy* (veja o Capítulo 16 para uma descrição desses protocolos).

- **Pinos de extensão do barramento para 64 bits:** inclui 32 linhas multiplexadas para endereços e dados, que são combinadas com linhas obrigatórias de endereço/dados para formar um barramento de endereços/dados de 64 bits. As demais linhas nesse grupo são utilizadas para interpretar e validar as linhas de sinais que carregam endereços e dados. Finalmente, existem duas linhas que possibilitam a dois dispositivos PCI entrar em acordo para uso do barramento estendido para 64 bits.
- **JTAG/pinos de teste:** essas linhas dão suporte aos procedimentos de teste definidos no Padrão IEEE 1149.1.

Comandos PCI

A atividade do barramento ocorre na forma de transações entre um iniciador, ou mestre, e um alvo. Quando um mestre adquire controle do barramento, ele determina o tipo de transação que ocorrerá em seguida. Durante a fase de endereço da transação, as linhas C/BE são empregadas para sinalizar o tipo da transação. Os possíveis comandos são:

- Confirmação de interrupção
- Ciclo Especial
- Leitura em dispositivo de E/S
- Escrita em dispositivo de E/S
- Leitura de memória
- Leitura de linha de memória
- Leitura múltipla de memória
- Escrita na memória
- Escrita na memória e invalidação
- Leitura de configuração
- Escrita de configuração
- Ciclo de endereço duplo

A Confirmação de Interrupção é um comando de leitura destinado ao dispositivo que funciona como controlador de interrupção no barramento PCI. As linhas de endereço não são utilizadas durante a fase de endereço e as linhas de habilitação de bytes de dados indicam o tamanho do identificador de interrupção a ser retornado.

O comando Ciclo Especial é utilizado pelo iniciador para difundir uma mensagem para um ou mais alvos.

Os comandos de Leitura e Escrita em dispositivos de E/S são empregados para transferir dados entre o iniciador e um controlador de E/S. Cada dispositivo de E/S tem seu próprio espaço de endereços; as linhas de endereço são utilizadas para indicar um dispositivo particular e para especificar os dados a serem transferidos de ou para esse dispositivo. O conceito de endereços de E/S é explorado no Capítulo 6.

Os comandos de leitura e escrita na memória são utilizados para especificar a transferência de uma porção de dados, com duração de um ou mais ciclos de relógio. A interpretação desses comandos depende de o controlador de memória no barramento PCI suportar ou não o protocolo PCI para transferências entre a memória e a memória cache. Em caso afirmativo, a unidade de transferência de dados de ou para a memória é tipicamente uma linha ou bloco da memória cache[2]. A utilização dos três comandos de leitura de memória é mostrado na Ta-

2 Os princípios fundamentais de memórias cache são descritos na Seção 4.3; os protocolos de cache em sistemas baseados em barramentos são descritos na Seção 16.3.

bela 3.5. O comando de Escrita na Memória é utilizado para transferir dados para a memória, em um ou mais ciclos. O comando Escrita na Memória e Invalidação transfere dados para a memória em um ou mais ciclos e, além disso, garante que pelo menos uma linha da memória cache seja escrita. Esse comando permite a implementação da função de escrever uma linha da memória cache de volta para a memória (técnica *write-back**).

Os dois comandos de configuração possibilitam ao mestre ler e atualizar parâmetros de configuração de um dispositivo conectado ao PCI. Cada dispositivo conectado ao PCI pode incluir até 256 registradores internos, utilizados para configurar o dispositivo durante a inicialização do sistema.

O comando Ciclo de Endereço Duplo é utilizado por um iniciador para indicar o uso de endereçamento de 64 bits.

Tabela 3.5 Interpretação dos comandos de leitura do PCI

Tipo de comando de leitura	Para memória cacheável	Para memória não-cacheável
Leitura de memória	Transferência de metade de uma linha de cache ou menos	Transferência de dados durante dois ciclos de relógio ou menos
Leitura de linha de memória	Transferência de mais da metade de uma linha a três linhas	Transferência de dados durante 3 a 12 ciclos de relógio
Leitura múltipla de memória	Transferência de mais de três linhas	Transferência de dados por mais de 12 ciclos de relógio

Transferências de dados

Toda transferência de dados no barramento PCI é uma transação única, consistindo em uma fase de transferência de endereço e uma ou mais fases de transferências de dados. Ilustramos, a seguir, uma operação de leitura típica; uma operação de escrita é feita de maneira análoga.

A Figura 3.22 mostra o diagrama de tempo para uma transação de leitura. Todos os eventos são sincronizados com as transições de queda (bordas de descida) do sinal de relógio, que ocorrem no meio de cada ciclo de relógio. Os dispositivos conectados ao barramento verificam os sinais nas linhas do barramento, em cada borda de subida no início de um ciclo de barramento. Os eventos indicados por rótulos no diagrama são descritos a seguir:

a. Uma vez que um mestre tenha obtido o controle do barramento, ele pode iniciar a transferência, ativando o sinal FRAME. Essa linha permanece ativada até que o iniciador esteja pronto para terminar a última fase de transferência de dados. O iniciador também coloca o endereço de início da transferência de dados no barramento de endereço, assim como coloca o comando de leitura nas linhas C/BE.

b. No início do segundo ciclo de relógio, o dispositivo-alvo deve reconhecer seu endereço nas linhas AD.

* N.R.T.: As técnicas de atualização dos dados modificados na memória cache serão estudadas mais adiante no Capítulo 4.

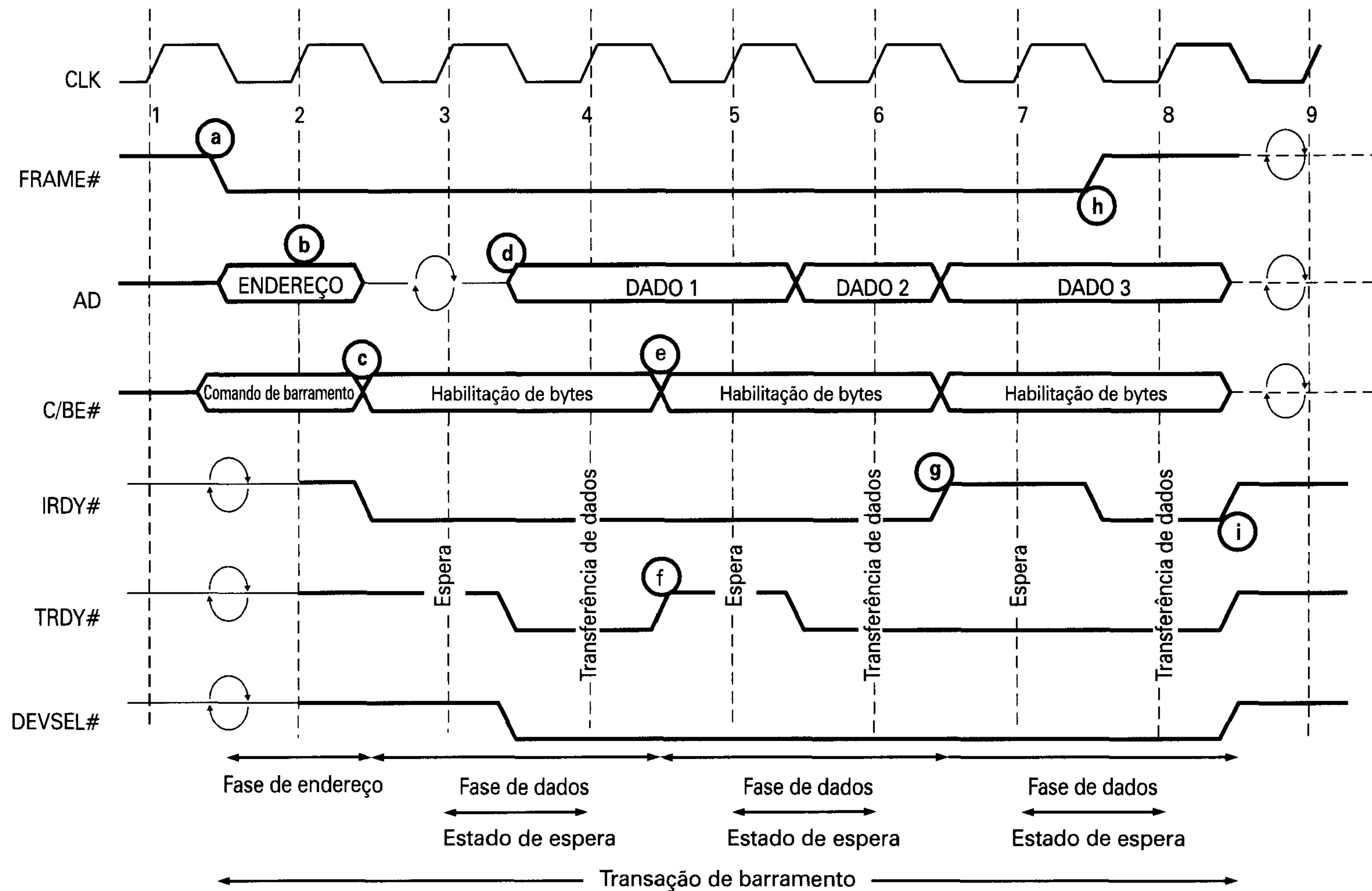


Figura 3.22 Operação de leitura no barramento PCI.

c. O iniciador pára de enviar informações nas linhas AD. É necessário um ciclo de espera (indicado no diagrama por duas setas circulares) em todas as linhas de sinal que possam ser utilizadas por mais de um dispositivo, de modo que a desativação do sinal de endereço irá preparar o barramento para ser usado pelo dispositivo-alvo. O iniciador muda a informação nas linhas C/BE para indicar que linhas AD serão utilizadas para transferência dos dados endereçados (de 1 a 4 bytes). Ele também ativa o sinal IRDY para indicar que está pronto para receber o primeiro item dos dados.
d. O dispositivo-alvo selecionado ativa o sinal DEVSEL para indicar que reconheceu seu endereço e que irá responder. Ele coloca os dados requisitados nas linhas AD e ativa o sinal TRDY para indicar que um dado válido está presente no barramento.
e. O iniciador lê os dados no início do quarto ciclo de relógio e muda os sinais das linhas de habilitação de bytes, como preparação para a próxima leitura.
f. Nesse exemplo, o dispositivo-alvo precisa de algum tempo para preparar o segundo bloco de dados para transmissão. Portanto, ele desativa o sinal TRDY para avisar ao iniciador que não enviará novos dados durante o próximo ciclo. Conseqüentemente, o iniciador não lê as linhas de dados no início do quinto ciclo de relógio nem muda o sinal de habilitação de bytes durante aquele ciclo. O novo bloco de dados é lido no início do sexto ciclo de relógio.
g. Durante o sexto ciclo de relógio, o dispositivo-alvo coloca o terceiro item de dados no barramento. Entretanto, nesse exemplo, o iniciador ainda não está pronto para ler o item de dados (por exemplo, sua área de armazenamento temporário está cheia). Portanto, ele desativa o sinal IRDY. Isso faz com que o alvo mantenha o terceiro item de dados no barramento por mais um ciclo de relógio.
h. O iniciador sabe que a terceira transferência de dados é a última e, portanto, desativa o sinal FRAME indicando ao alvo que essa é a última transferência. Além disso, ele ativa o sinal IRDY para indicar que está pronto para terminar a transferência.
i. O iniciador desativa o sinal IRDY, retornando o barramento para o estado inativo, e o dispositivo-alvo desativa os sinais TRDY e DEVSEL.

Arbitração

O barramento PCI utiliza um esquema de arbitração síncrono e centralizado, no qual cada mestre possui uma linha de sinal de requisição (REQ) e uma linha de sinal de concessão (GNT). Essas linhas são conectadas a um árbitro central (Figura 3.23) e um protocolo simples de requisição/concessão é empregado para obter acesso ao barramento.

A especificação do PCI não determina um algoritmo de arbitração particular. O árbitro do barramento pode utilizar uma abordagem de ceder o barramento segundo a ordem de chegada das requisições (a primeira a chegar é a primeira a ser atendida), ou uma abordagem de compartilhamento circular de tempo (*round-robin*), ou algum tipo de esquema de prioridade. Cada mestre PCI deve requisitar o controle do barramento para cada transação que deseje efetuar, em que cada transação seja composta de uma fase de transferência de endereço seguida por uma ou mais fases contíguas de transferências de dados.

A Figura 3.24 ilustra um exemplo no qual o controle do barramento é arbitrado entre dois dispositivos, *A* e *B*. Ocorre a seguinte seqüência de eventos:

a. Em algum ponto anterior ao início do ciclo de relógio 1, o dispositivo *A* ativa seu sinal REQ. O árbitro verifica esse sinal no início do ciclo de relógio 1.

b. Durante o ciclo de relógio 1, o dispositivo *B* requisita uso do barramento, ativando o seu sinal REQ.

c. Ao mesmo tempo, o árbitro ativa o sinal GNT do dispositivo *A* (GNT-A), cedendo a esse dispositivo o controle do barramento.

d. O mestre de barramento *A* verifica o sinal na sua linha GNT (GNT-A), no início do ciclo de relógio 2, e toma conhecimento de que obteve o controle do barramento. Ele verifica também que não existe sinal nas linhas IRDY e TRDY, o que indica que o barramento está ocioso. Conseqüentemente, ativa o sinal FRAME e coloca a informação de endereço no barramento de endereço e o comando no barramento C/BE (não mostrado na figura). Além disso, mantém seu sinal REQ (REQ-A), uma vez que tem uma segunda transação a efetuar em seguida.

e. O árbitro do barramento testa todas as linhas GNT, no início do ciclo de relógio 3, e decide ceder o controle do barramento ao dispositivo *B* para a próxima transação. Então, ele ativa o sinal GNT do dispositivo *B* (GNT-B) e desativa o sinal GNT do dispositivo *A* (GNT-A). O dispositivo *B* não poderá usar o barramento até que ele retorne ao estado ocioso.

f. O dispositivo *A* desativa o sinal FRAME, indicando que a última (e única) transferência de dados está em andamento. Ele coloca os dados no barramento de dados e envia o sinal IRDY para o alvo. O alvo lê os dados no início do ciclo de relógio seguinte.

g. No início do ciclo de relógio 5, o dispositivo *B* verifica que os sinais IRDY e FRAME estão desativados e que, portanto, ele pode tomar o controle do barramento ativando o sinal FRAME. Além disso, ele desativa o sinal na sua linha REQ, uma vez que deseja efetuar apenas uma transação.

Em seguida, o mestre *A* obtém o controle do barramento para sua próxima transação.

Note que a arbitração do barramento pode ocorrer ao mesmo tempo em que o mestre corrente do barramento está efetuando uma transferência de dados. Portanto, não há perda de ciclos de barramento por causa do processo de arbitração. Esse processo é conhecido como *arbitração oculta*.

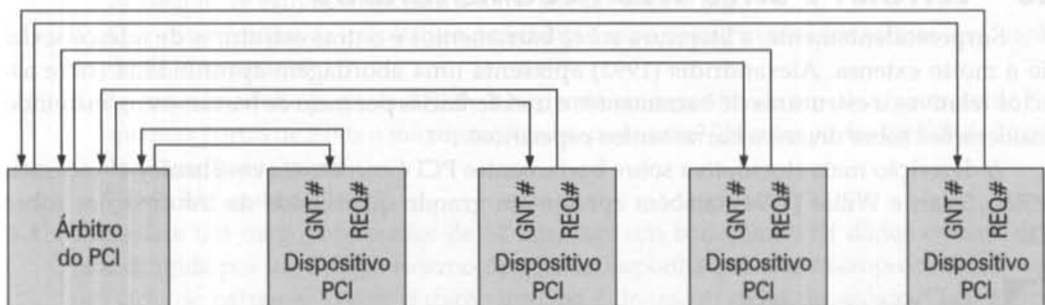


Figura 3.23 Árbitro de barramento PCI.

Figura 3.24 Arbitração de barramento PCI entre dois mestres.

3.6 LEITURA E SITES WEB RECOMENDADOS

Surpreendentemente, a literatura sobre barramentos e outras estruturas de interconexão não é muito extensa. Alexandridis (1993) apresenta uma abordagem aprofundada sobre aspectos relativos a estruturas de barramentos e transferências por meio de barramentos, incluindo considerações sobre diversos barramentos específicos.

A descrição mais elucidativa sobre barramentos PCI é encontrada em Shanley e Anderson (1995b). Solari e Willse (1994) também apresentam grande quantidade de informações sobre PCI.



Site Web recomendado:

- **PCI Special Interest Group:** informações sobre especificações e produtos PCI.

3.7 EXERCÍCIOS

3.1 A máquina hipotética da Figura 3.4 possui também duas instruções de E/S:

0011 = carregar AC a partir de um dispositivo de E/S

0111 = armazenar AC em um dispositivo de E/S

Nessas instruções, o endereço de 12 bits identifica um dispositivo de E/S particular. Descreva a execução do seguinte programa (utilizando o formato apresentado na Figura 3.5):

1. Carregar AC a partir do dispositivo 5.
2. Somar com o conteúdo da posição de memória 940.
3. Armazenar o conteúdo de AC no dispositivo 6.

Suponha que o próximo valor obtido do dispositivo 5 seja 3 e que a posição de memória 940 contenha o valor 2.

3.2 Considere um microprocessador hipotético de 32 bits, cujas instruções de 32 bits são compostas de dois campos: o primeiro byte contém o código de operação e os demais contêm um operando imediato ou um endereço de operando.

a. Qual é a capacidade máxima de memória endereçável diretamente (em bytes)?

b. Discuta qual o impacto sobre velocidade do sistema, caso o barramento do microprocessador tenha:

1. um barramento local de endereços de 32 bits e um barramento local de dados de 16 bits; ou
2. um barramento local de endereços de 16 bits e um barramento local de dados de 16 bits.

c. Quantos bits são necessários para o contador de programa e para o registrador de instrução?

Fonte: Alexandridis (1993).

3.3 Considere um microprocessador hipotético que gera um endereço de 16 bits (suponha, por exemplo, que o contador de programa e os registradores de endereço tenham 16 bits) e que possua um barramento de dados de 16 bits.

a. Qual é o maior espaço de endereçamento à memória que o processador pode acessar diretamente, se estiver conectado a uma "memória de 16 bits"?

b. Qual é o maior espaço de endereçamento à memória que o processador pode acessar diretamente, se estiver conectado a uma "memória de 8 bits"?

c. Que característica da arquitetura possibilita a esse microprocessador acessar um "espaço de E/S" separado?

d. Se um número de porta de E/S de 8 bits pode ser especificado em uma instrução de E/S, quantas portas de 8 bits o microprocessador pode usar? Quantas portas de E/S de 16 bits? Explique sua resposta.

Fonte: Alexandridis (1993).

3.4 Considere um microprocessador de 32 bits, com um barramento de dados externo de 16 bits, dirigido por um relógio externo de 8 MHz. Suponha que esse microprocessador tenha um ciclo de barramento cuja duração mínima é de quatro ciclos de relógio. Qual é a taxa máxima de transferência de dados que esse microprocessador pode sustentar? Para aumentar seu desempenho, seria melhor aumentar a largura do seu barramento de dados externo de 16 para 32 bits ou dobrar a freqüência do relógio externo fornecido ao microprocessador? Enuncie qualquer suposição que você precise fazer e explique sua resposta.

Fonte: Alexandridis (1993).

3.5 Considere um sistema de computação que contenha um módulo de E/S que controla um teletipo com teclado/impressora padrão. O processador inclui os seguintes registradores, diretamente conectados ao barramento do sistema:

INPR: registrador de entrada, 8 bits
OUTR: registrador de saída, 8 bits
FGI: indicador de entrada, 1 bit
FGO: indicador de saída, 1 bit
IEN: habilitação de interrupção, 1 bit

A entrada de uma tecla a partir do teletipo, assim como a saída impressa no teletipo, é controlada pelo módulo de E/S. O teletipo é capaz de codificar um símbolo alfanumérico em uma palavra de 8 bits e de decodificar uma palavra de 8 bits em um símbolo alfanumérico.

a. Descreva como o processador pode realizar uma operação de E/S com o teletipo, usando os quatro primeiros registradores relacionados.

b. Descreva como essa operação pode ser efetuada de maneira mais eficiente, empregando também o registrador IEN.

3.6 A Figura 3.25 mostra um esquema de arbitração distribuído que pode ser usado com o barramento Multibus I. Os agentes são fisicamente encadeados em ordem de prioridade. O agente mais à esquerda no diagrama recebe um sinal constante na linha de *entrada de prioridade de barramento* (BPRN), indicando que nenhum agente de maior prioridade está requisitando o barramento. Caso esse agente não deseje obter o controle do barramento, ele ativa sua linha BPRO (*saída de prioridade de barramento*). No início de cada ciclo de relógio, qualquer agente pode requisitar o controle do barramento desativando a sua linha BPRO. Isso desativa a linha BPRN do agente seguinte na cadeia, que, por sua vez, deve desativar sua linha BPRO. Assim, o sinal é propagado ao longo da cadeia. Ao final dessa reação em cadeia, deve haver apenas um agente cujo sinal BPRN está ativado e cujo BPRO não está ativado. Esse agente tem prioridade para obter o controle do barramento. Se no início do ciclo o barramento não estiver ocupado (sinal BUSY inativo), o agente que tiver prioridade poderá obter o controle do barramento ativando a linha BUSY (ocupado).

O sinal BPR leva certo tempo para se propagar do agente de maior prioridade para o de menor prioridade. Esse tempo deve ser menor do que o ciclo de relógio? Explique.

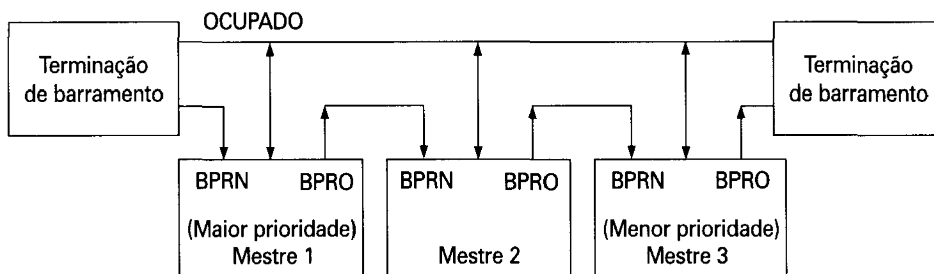


Figura 3.25 Arbitração distribuída do barramento Multibus I.

3.7 O barramento SBI do VAX utiliza um esquema de arbitração distribuído e síncrono. Cada dispositivo SBI (por exemplo, um processador, uma memória ou um adaptador de barramento Unibus) tem uma prioridade diferente e uma linha de requisição de transferência (*transfer request* — TR) própria. O SBI possui 16 dessas linhas (TR0, TR1, ..., TR15), sendo a linha TR0 a de prioridade mais alta. Quando um dispositivo deseja usar o barramento, ele faz uma reserva para uso do barramento em um intervalo de tempo futuro, ativando sua linha TR durante o intervalo de tempo atual. Ao final do intervalo de tempo atual, cada dispositivo que tem uma reserva pendente examina as linhas TR; o dispositivo com reserva que tiver maior prioridade usará o barramento no próximo intervalo de tempo.

No máximo 17 dispositivos podem ser conectados ao barramento. O dispositivo com prioridade 16 não possui uma linha TR. Por quê?

3.8 Paradoxalmente, o dispositivo de menor prioridade geralmente tem o menor tempo médio de espera. Por essa razão, o processador tem normalmente a menor prioridade no SBI. Por que o dispositivo de prioridade 16 geralmente tem o menor tempo médio de espera? Em que circunstâncias isso não seria verdadeiro?

3.9 Desenhe e explique um diagrama de tempo para uma operação de escrita no PCI (semelhante à Figura 3.22).

APÊNDICE 3A DIAGRAMAS DE TEMPO

Neste capítulo, diagramas de tempo são utilizados para mostrar seqüências de eventos e dependências entre os eventos. Este apêndice apresenta uma breve descrição de diagramas de tempo para o leitor que não esteja familiarizado com eles.

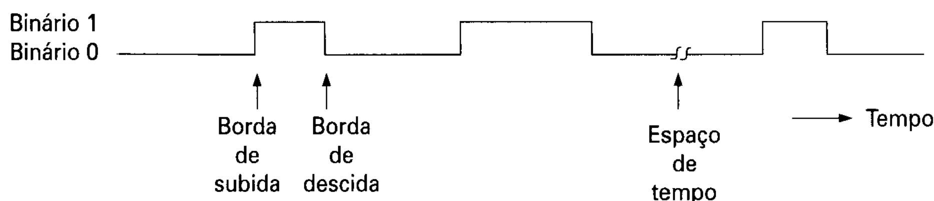
A comunicação entre os dispositivos conectados a um barramento ocorre por meio de um conjunto de linhas capazes de carregar sinais. Dois níveis distintos de sinais (níveis de voltagem), representando os dígitos binários 0 e 1, podem ser transmitidos. Um diagrama de tempo mostra o nível de sinal em uma linha, em função do tempo (Figura 3.26a). Por convenção, o nível de sinal correspondente ao dígito 1 é representado como um nível mais alto do que o nível de sinal correspondente ao dígito 0. Geralmente, o dígito 0 é o valor padrão, isto é, se nenhum sinal ou valor estiver sendo transmitido, então o nível de sinal na linha é aquele que representa o dígito 0. Uma transição de um sinal de 0 para 1 é freqüentemente denominada *borda de subida* do sinal; uma transição de 1 para 0 é denominada *borda de descida*. Para maior clareza, as transições de sinal são sempre representadas como se ocorressem instantaneamente. Na verdade, uma transição leva um intervalo de tempo não-nulo, mas esse tempo de transição geralmente é muito menor do que a duração de um nível de sinal. Em um diagrama de tempo, um intervalo de tempo variável, ou pelo menos um intervalo irrelevante, pode transcorrer entre eventos de interesse. Isso é representado por um espaço (*gap*) na linha de tempo.

Os sinais algumas vezes são representados em grupos (Figura 3.26b). Por exemplo, se a transferência de dados é feita em unidades de byte, oito linhas de dados são requeridas. Geralmente não é importante saber o valor exato que está sendo transferido nesse grupo de linhas, mas apenas se sinais válidos estão presentes ou não nessas linhas.

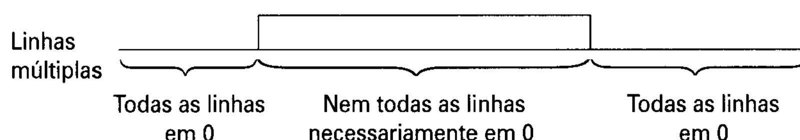
Uma transição de sinal em uma linha pode fazer com que um dispositivo a ela conectado dispare mudanças de sinal em outras linhas. Por exemplo, se um módulo de memória detecta um sinal de controle de leitura (transição 0 ou 1), ele coloca sinais de dados nas linhas de

dados. Essa relação de causa e efeito produz uma seqüência de eventos. Nos diagramas de tempo, essas dependências entre eventos são representadas por setas (Figura 3.26c).

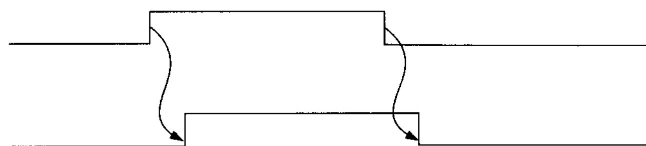
Freqüentemente, o barramento do sistema inclui uma linha de relógio. Um relógio eletrônico é conectado à linha de relógio e fornece uma seqüência repetitiva e regular de transições (Figura 3.26d). Outros eventos podem ser sincronizados ao sinal de relógio.



(a) Sinal como uma função do tempo



(b) Grupos de linhas



(c) Dependências de causa e efeito

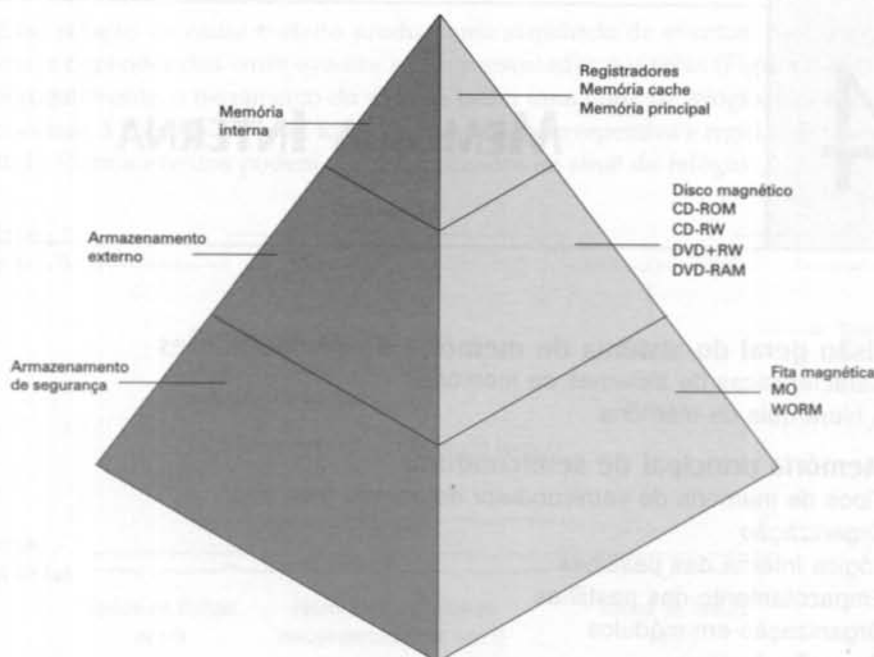


(d) Sinal de relógio

Figura 3.26 Diagramas de tempo.

capítulo

4 Memória Interna



- A memória de um computador é organizada de maneira hierárquica. O nível superior (mais próximo do processador) é constituído de registradores do processador. Em seguida, vem um ou dois níveis de memória cache, designados como caches *L1* e *L2*. Depois vem a memória principal, que normalmente usa módulos de memória dinâmica de acesso aleatório (*dynamic random-access memory* — DRAM). Essas memórias são consideradas internas ao sistema de computação. A hierarquia continua com a memória externa, na qual o nível seguinte é tipicamente composto por um disco rígido fixo, e com os níveis abaixo constituídos de meios removíveis, tais como cartuchos ZIP, discos ópticos e fitas magnéticas.
- À medida que descemos pela hierarquia de memória, o custo por bit torna-se menor, a capacidade de memória fica maior e o tempo de acesso, mais lento. O ideal seria usar apenas a memória mais rápida; entretanto, como essas memórias são as mais caras, o tempo de acesso é sacrificado em favor de um custo mais baixo, utilizando memórias mais lentas. A idéia é organizar dados e programas na memória de maneira que as palavras de memória requeridas geralmente sejam encontradas nas memórias mais rápidas.
- Em geral, é provável que a maioria dos acessos futuros à memória principal pelo processador sejam as posições de memória usadas recentemente. Assim, a memória cache mantém automaticamente uma cópia de algumas palavras da memória DRAM usadas recentemente. Com um projeto adequado, as palavras requisitadas pelo processador estarão, na maioria das vezes, armazenadas na memória cache.

Embora o conceito de memória seja aparentemente simples, é talvez aquele componente que apresenta maior variedade de tipos, tecnologias, organizações, desempenhos e custos, entre todos os elementos de um sistema de computador. Nenhuma das tecnologias de memória existentes satisfaz de maneira ótima todos os requisitos de armazenamento de dados em computadores. Assim, um sistema de computador típico é equipado com uma hierarquia de subsistemas de memória, algumas internas (diretamente acessíveis pelo processador) e outras externas (acessíveis pelo processador por meio de um módulo de E/S).

Este capítulo enfatiza a descrição de memórias internas, enquanto as memórias externas são tratadas no Capítulo 5. A primeira seção aborda as características fundamentais dos sistemas de memória dos computadores. As seções seguintes examinam os subsistemas de memória principal de semicondutor, incluindo memórias ROM, DRAM e SRAM. Em seguida, um elemento essencial dos sistemas de computação modernos é discutido: a memória cache. Depois disso, estamos prontos para retornar à questão das memórias DRAM e examinar algumas arquiteturas mais avançadas.

4.1 VISÃO GERAL DO SISTEMA DE MEMÓRIA DE COMPUTADORES

Características de sistemas de memória

Os sistemas de memória de computadores podem ser mais facilmente compreendidos por meio de sua classificação, de acordo com suas características fundamentais. As características mais importantes são relacionadas na Tabela 4.1.

Na Tabela 4.1, o termo **localização** é empregado para indicar se a memória é interna ou externa ao computador. Embora a memória interna seja geralmente identificada com a memória principal, existem outras formas de memória interna. O processador requer uma memória local própria, formada por registradores (veja, por exemplo, a Figura 2.3). Além disso, como veremos no decorrer deste capítulo, a unidade de controle do processador pode também ter sua própria memória interna. Esses dois tipos de memória interna serão discutidos em capítulos posteriores. Outro tipo de memória interna é a memória cache. A memória externa consiste em dispositivos de armazenamento periféricos, tais como discos e fitas, que são acessíveis ao processador por meio de controladores de E/S.

Tabela 4.1 Características fundamentais de sistemas de memória de computadores

Localização	**Desempenho**
Processador	Tempo de acesso
Interna (principal)	Tempo de ciclo
Externa (secundária)	Taxa de transferência
Capacidade	**Tecnologia**
Tamanho da palavra	De semicondutores
Número de palavras	Magnética
Unidade de transferência	Óptica
Palavra	Magneto-óptica
Bloco	**Características físicas**
Método de acesso	Volátil/não-volátil
Seqüencial	Apagável/não-apagável
Direto	
Aleatório	**Organização**
Associativo	

Uma característica óbvia de uma memória é sua **capacidade**. Na memória interna, a capacidade é usualmente expressa em função de bytes (1 byte = 8 bits) ou palavras. Os tamanhos mais usuais de palavras são 8, 16 e 32 bits. Na memória externa, a capacidade é tipicamente expressa em função de bytes.

Um conceito relacionado é a **unidade de transferência de dados**. Na memória interna, a unidade de transferência é igual ao número de linhas de dados do módulo de memória. Embora esse número de linhas seja freqüentemente igual ao tamanho da palavra, isso não é necessário. Para entender por que, considere os três conceitos relacionados à memória interna a seguir:

- **Palavra:** unidade "natural" de organização da memória. O tamanho de uma palavra é tipicamente igual ao número de bits usados para representar um número inteiro e ao tamanho de uma instrução. Infelizmente, há muitas exceções. Por exemplo, no CRAY 1 uma palavra tem 64 bits, mas a representação de números inteiros utiliza 24 bits. No VAX uma palavra tem 32 bits e existe uma enorme variedade de tamanhos de instruções, expressos como múltiplos de byte.
- **Unidade endereçável:** em muitos sistemas, a unidade endereçável de dados é a palavra. Entretanto, alguns sistemas permitem o endereçamento de bytes. Em qualquer um dos casos, a relação entre o tamanho em bits A de um endereço e o número de unidades endereçáveis N é $2^A = N$.
- **Unidade de transferência:** a unidade de transferência de dados da memória principal é o número de bits que podem ser lidos ou escritos de cada vez. Ela não precisa ser igual a uma palavra ou à unidade endereçável de dados. Na memória externa, os dados são freqüentemente transferidos em unidades muito maiores do que uma palavra, chamadas de blocos.

Outra forma de diferenciação entre tipos de memória é o **método de acesso** aos dados, que pode ser:

- **Acesso seqüencial:** os dados são organizados na memória em unidades chamadas registros. O acesso é feito segundo uma seqüência linear específica. Além dos dados, são armazenadas informações de endereçamento, utilizadas para separar um registro do registro seguinte e facilitar o processo de busca por um determinado registro. Um mecanismo compartilhado é usado para leitura e escrita; a cada operação ele deve ser movido de sua posição atual para a desejada, ignorando registros intermediários. Portanto, o tempo de acesso a um registro arbitrário varia muito. As unidades de fitas, discutidas no Capítulo 5, são dispositivos de memória de acesso seqüencial.
- **Acesso direto:** assim como com o acesso seqüencial, o acesso direto emprega um mecanismo compartilhado para leitura e escrita. Entretanto, cada bloco individual ou registro possui um endereço único, baseado em sua localização física. O acesso é feito por meio de um acesso direto a uma vizinhança genérica do registro e, em seguida, por uma pesquisa seqüencial, por contagem ou por espera até atingir a posição final. O tempo de acesso também é variável. As unidades de disco, discutidas no Capítulo 5, são dispositivos de memória de acesso direto.
- **Acesso aleatório:** cada posição de memória endereçável possui um mecanismo de endereçamento único e fisicamente conectado a ela. O tempo de acesso a uma determinada posição é constante e independente da seqüência de acessos anteriores. Dessa maneira, qualquer posição pode ser selecionada de modo aleatório, sendo endereçada e acessada diretamente. A memória principal, assim como alguns sistemas de memória cache, é um dispositivo de memória de acesso aleatório.
- **Associativo:** consiste em um tipo de memória de acesso aleatório que possibilita comparar simultaneamente certo número de bits de uma palavra com todas as palavras da memória, determinando quais dessas palavras contêm o mesmo padrão de bits. Uma palavra é buscada na memória com base em uma parte do seu conteúdo, e não de acordo com seu endereço. Assim como na memória de acesso aleatório, cada posição da memória possui seu mecanismo de endereçamento próprio e o tempo de busca é constante e independente da posição ou do padrão dos acessos anteriores. As memórias cache, discutidas na Seção 4.3, podem empregar acesso associativo.

Do ponto de vista do usuário, as duas características mais importantes da memória são sua capacidade e seu **desempenho**. Os parâmetros empregados para medir o desempenho são:

- **Tempo de acesso:** em uma memória de acesso aleatório, esse é o tempo gasto para efetuar uma operação de leitura ou de escrita: é o tempo decorrido desde o instante em que um endereço é apresentado à memória até o momento em que os dados são armazenados ou se tornam disponíveis para utilização. Em uma memória de acesso não-aleatório, o tempo de acesso é o tempo gasto para posicionar o mecanismo de leitura-escrita na posição desejada.
- **Tempo de ciclo de memória:** esse conceito é aplicável principalmente a memórias de acesso aleatório e compreende o tempo de acesso e o tempo adicional requerido antes que um segundo acesso possa ser iniciado. Esse tempo adicional pode ser necessário para o desaparecimento de transientes nas linhas de sinais ou para a regeneração dos dados, caso a leitura seja destrutiva.

- **Taxa de transferência:** é a taxa na qual os dados podem ser transferidos de ou para a unidade de memória. Na memória de acesso aleatório, é equivalente a 1/(tempo de ciclo).

Para uma memória de acesso não-aleatório, é válida a seguinte relação:

$$T_N = T_A + \frac{N}{R}$$

onde:

T_N = tempo médio para ler ou escrever N bits
T_A = tempo médio de acesso
N = número de bits
R = taxa de transferência em bits por segundo (bps)

Diversas **tecnologias** têm sido empregadas para a fabricação de memórias de computadores. As mais comuns atualmente são as memórias de semicondutor, as memórias de superfície magnética, utilizadas em discos e fitas, e as memórias ópticas e magneto-ópticas.

Diversas **características físicas** de armazenamento são importantes. Em uma memória volátil, os dados são perdidos quando a energia elétrica é desligada. Em uma memória não-volátil, os dados, uma vez gravados, permanecem armazenados sem alteração até serem explicitamente modificados; nenhuma energia é requerida para manter os dados armazenados. As memórias de superfície magnética são não-voláteis. As memórias de semicondutor podem ser tanto voláteis quanto não-voláteis. O conteúdo de memórias não-apagáveis não pode ser alterado, a menos que se destrua a unidade de armazenamento. As memórias de semicondutor desse tipo são denominadas *memória apenas de leitura* (*read-only memory* — ROM). Para que tenha alguma utilidade, uma memória não-apagável deve ser também não-volátil.

A **organização** é um aspecto fundamental do projeto de memórias de acesso aleatório. Por organização entende-se o arranjo físico dos bits para formar palavras. Nem sempre um arranjo óbvio é usado, como será explicado a seguir.

A hierarquia de memória

As restrições de projeto de uma memória podem ser resumidas em três questões: capacidade, velocidade e custo.

A questão referente à capacidade é, de certo modo, indefinida. Qualquer que seja a capacidade disponível, provavelmente serão desenvolvidas novas aplicações que a utilizem integralmente. A questão relativa à velocidade tem, de certa maneira, uma resposta mais fácil. Para obter um melhor desempenho, a velocidade da memória deve ser compatível com a do processador. Ou seja, o processador não deve ficar ocioso esperando que instruções ou operandos sejam buscados na memória durante a execução de instruções. A questão sobre o custo também deve ser considerada. Para que um sistema seja comercialmente viável, o custo da memória deve ser compatível com o dos demais componentes.

Como se poderia esperar, as três características principais da memória — custo, capacidade e tempo de acesso — são conflitantes. Uma variedade de tecnologias é utilizada para a implementação de sistemas de memória. Ao longo desse espectro de tecnologias, valem as seguintes relações:

- Tempo de acesso mais rápido, custo por bit maior.
- Capacidade maior, custo por bit menor.
- Capacidade maior, tempo de acesso menor.

O dilema com o qual se depara um projetista é claro. Seria desejável usar uma tecnologia de memória capaz de fornecer uma grande capacidade de armazenamento de dados, porque uma grande capacidade é necessária e o custo por bit é mais baixo. Entretanto, para obter um desempenho melhor, o projetista precisa utilizar memórias caras, que apresentam tempo de acesso menor, mas com capacidade relativamente mais baixa.

A saída para esse dilema é empregar uma **hierarquia de memórias**, e não um único componente ou tecnologia de memória. Uma hierarquia típica é mostrada na Figura 4.1. À medida que descemos em uma hierarquia de memórias, as relações a seguir são válidas:

a. O custo por bit diminui.
b. A capacidade aumenta.
c. O tempo de acesso aumenta.
d. A freqüência de acesso à memória pelo processador diminui.

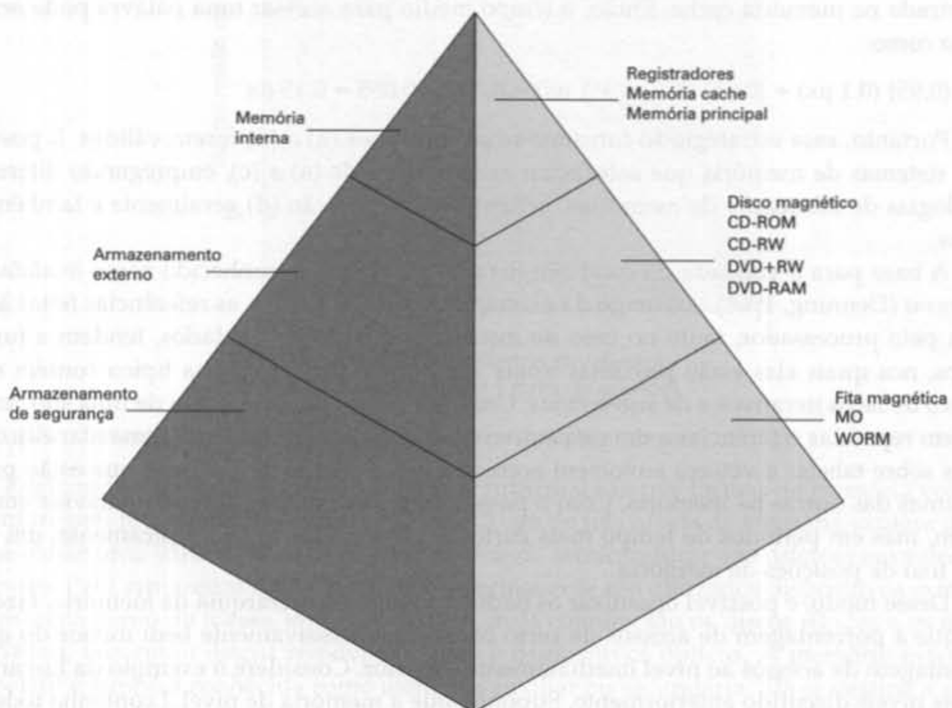


Figura 4.1 Hierarquia de sistemas de memória.

Desse modo, memórias menores, mais caras e mais rápidas são combinadas com memórias maiores, mais baratas e mais lentas. A chave para o sucesso dessa organização está no item (d): a diminuição da freqüência de acessos. Esse conceito é examinado mais detalhadamente ao final deste capítulo, quando discutimos o uso de memórias cache, e no Capítulo 7, no qual abordamos o mecanismo de memória virtual. Uma breve explicação é dada a seguir.

Considere um processador com acesso a dois níveis de memória. O nível 1 contém mil palavras e o tempo de acesso é de 0,1 μs; o nível 2 contém 100 mil palavras e o tempo de acesso é de 1 μs. Suponha que, se a palavra requerida está no nível 1, ela é acessada diretamente pelo processador. Mas, caso esteja no nível 2, ela tem de ser primeiramente transferida para o nível 1 para depois ser acessada pelo processador. Para maior simplicidade, ignoramos o tempo requerido para o processador determinar se a palavra está no nível 1 ou no nível 2. A Figura 4.2 mostra a forma geral da curva que corresponde a essa situação. Ela ilustra um gráfico do tempo médio de acesso à memória de dois níveis, em função da taxa de acertos *H*, onde *H* é definido como a fração de acessos à memória em que a palavra requerida é encontrada na memória flash (por exemplo, a memória cache); T_1 é o tempo de acesso ao nível 1 e T_2, o tempo de acesso ao nível 2. Como se pode observar, se a taxa de acessos ao nível 1 é alta, então o tempo total de acesso médio é muito mais próximo do tempo de acesso ao nível 1 do que do tempo de acesso ao nível 2.

Suponha, nesse exemplo, que em 95% dos acessos à memória a palavra requerida seja encontrada na memória cache. Então, o tempo médio para acessar uma palavra pode ser expresso como:

$$(0{,}95)\ (0{,}1\ \mu s) + (0{,}05)\ (0{,}1\ \mu s + 1\ \mu s) = 0{,}095 + 0{,}055 = 0{,}15\ \mu s$$

Portanto, essa estratégia só funciona se as condições (a) a (d) forem válidas. É possível obter sistemas de memória que satisfaçam as condições de (a) a (c), empregando diferentes tecnologias de fabricação de memórias. Felizmente, a condição (d) geralmente é também satisfeita.

A base para a validade da condição (d) está no princípio conhecido como *localidade de referências* (Denning, 1968). Ao longo da execução de um programa, as referências feitas à memória pelo processador, tanto no caso de instruções quanto no de dados, tendem a formar grupos, nos quais elas estão próximas umas das outras. Um programa típico contém certo número de laços iterativos e de sub-rotinas. Uma vez dentro de um laço ou de uma sub-rotina, ocorrem repetidas referências a um pequeno conjunto de instruções. Da mesma maneira, operações sobre tabelas e vetores envolvem acessos a um conjunto de palavras que estão próximas umas das outras na memória. Com o passar do tempo, os conjuntos de palavras em uso variam, mas em períodos de tempo mais curtos o processador utiliza, tipicamente, um conjunto fixo de posições de memória.

Desse modo, é possível organizar os dados ao longo da hierarquia de memória, fazendo com que a porcentagem de acessos de certo nível seja sucessivamente bem menor do que a porcentagem de acessos ao nível imediatamente superior. Considere o exemplo da hierarquia de dois níveis discutido anteriormente. Suponha que a memória de nível 2 contenha todos os dados e instruções do programa. Os conjuntos de dados mais utilizados em cada instante podem ser temporariamente armazenados no nível 1. De tempos em tempos, alguns desses conjuntos de dados têm de ser removidos de volta para o nível 2, para dar espaço a um novo conjunto de dados no nível 1. Entretanto, em média, a maioria das referências é feita para instruções e dados que estão no nível 1.

Esse princípio pode ser aplicado a mais de dois níveis de memória, como sugere a hierarquia mostrada na Figura 4.1. A memória flash, menor e mais cara, é constituída de registradores internos do processador. Um processador típico contém algumas dezenas de registradores internos, embora algumas máquinas possuam centenas de registradores. A me-

mória principal, dois níveis abaixo, é o sistema de memória interna mais importante do computador. Cada posição da memória principal tem um endereço único; a maioria das instruções de máquina refere-se a um ou mais endereços da memória principal. A memória principal geralmente é combinada com uma memória cache menor e de maior velocidade. A memória cache normalmente não é visível para o programador ou mesmo para o processador. Ela é apenas um dispositivo que organiza a movimentação de dados entre a memória principal e os registradores do processador, de modo que melhore o desempenho.

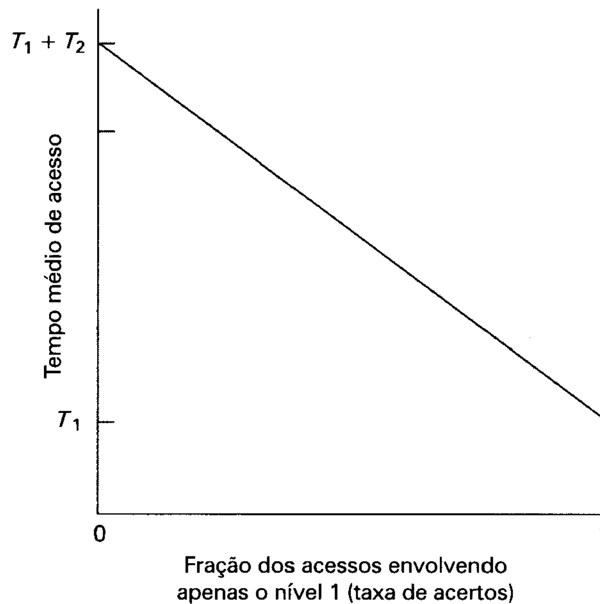


Figura 4.2 Desempenho de uma memória simples com dois níveis.

Registradores, memória cache e memória principal são três formas de memória voláteis que empregam tecnologia de semicondutores. O uso de três níveis de memória explora o fato de que existe uma variedade de tipos de memória de semicondutor que diferem em velocidade e custo. Para armazenar dados de maneira permanente são utilizados dispositivos externos de armazenamento de massa, entre os quais os mais comuns são os discos rígidos e os meios removíveis, tais como discos removíveis, fitas e dispositivos ópticos. A memória externa e não-volátil é também conhecida como memória auxiliar ou secundária. Ela se destina a armazenar programas e arquivos de dados e em geral é visível ao programador em termos de arquivos e registros, e não em termos de palavras ou bytes. Discos são também utilizados como uma extensão da memória principal conhecida como memória virtual, que será discutida no Capítulo 7.

Outras formas de memória podem ser incluídas na hierarquia. Por exemplo, computadores de grande porte da IBM introduzem uma forma de memória interna conhecida como memória expandida. Essa memória utiliza uma tecnologia de semicondutores mais lenta e mais barata do que a empregada na memória principal. Estritamente falando, ela não se encaixa na hierarquia, sendo um ramo lateral: os dados podem ser transferidos entre a memória principal e a memória expandida, mas não entre a memória expandida e a memória externa.

Outras formas de memória secundária incluem discos ópticos e magneto-ópticos. Finalmente, podem ser introduzidos níveis adicionais na hierarquia, por meio do uso de software. Uma parte da memória principal pode ser utilizada como uma área de armazenamento temporário de dados a serem gravados no disco. Essa técnica, algumas vezes denominada cache de disco[1], contribui para melhorar o desempenho do sistema de duas maneiras:

- As operações de escrita em disco são agrupadas. Em vez de realizar várias transferências de pequenas quantidades de dados, são feitas poucas transferências de grandes blocos. Isso melhora o desempenho dos discos e minimiza a utilização do processador.
- Alguns dados a serem escritos em um disco podem ser referenciados novamente pelo programa antes que seja feita a próxima transferência de dados da cache para esse disco. Nesse caso, os dados são obtidos mais rapidamente da cache de disco do que do disco físico.

O Apêndice 4A examina as implicações no desempenho de um sistema do uso de uma estrutura de memória de múltiplos níveis.

4.2 MEMÓRIA PRINCIPAL DE SEMICONDUTOR

Nos primeiros computadores, a tecnologia mais comumente utilizada para fabricação da memória principal de acesso aleatório empregava um grupo de anéis de material ferromagnético, conhecidos como *núcleos*. Desde seu aparecimento, as memórias baseadas na microeletrônica superaram de longe as memórias de núcleo magnético. Atualmente, o uso de pastilhas de semicondutores para a memória principal é quase universal. Os aspectos fundamentais dessa tecnologia são abordados nesta seção.

Tabela 4.2 Tipos de memória de semicondutor

<table>
<tr><th>Tipo de memória</th><th>Categoria</th><th>Mecanismo de apagamento</th><th>Mecanismo de escrita</th><th>Volatilidade</th></tr>
<tr><td>Memória de acesso aleatório (RAM)</td><td>Memória de leitura e de escrita</td><td>Eletricamente, em nível de bytes</td><td>Eletricamente</td><td>Volátil</td></tr>
<tr><td>Memória apenas de leitura (ROM)</td><td rowspan="2">Memória apenas de leitura</td><td rowspan="2">Não é possível</td><td>Máscaras</td><td rowspan="5">Não-volátil</td></tr>
<tr><td>ROM programável (PROM)</td><td rowspan="4">Eletricamente</td></tr>
<tr><td>PROM apagável (EPROM)</td><td rowspan="3">Memória principalmente de leitura</td><td>Luz UV, em nível de pastilha</td></tr>
<tr><td>Memória flash</td><td>Eletricamente, em nível de blocos</td></tr>
<tr><td>PROM eletricamente apagável (EEPROM)</td><td>Eletricamente, em nível de bytes</td></tr>
</table>

1 Em geral, a técnica de cache de disco é realizada puramente por software e não é examinada neste livro. Veja Stallings (1998) para uma abordagem sobre o assunto.

Tipos de memória de semicondutor de acesso aleatório

Todos os tipos de memória examinados nesta seção são de acesso aleatório, isto é, palavras individuais de memória são acessadas diretamente, utilizando uma lógica de endereçamento implementada em hardware.

A Tabela 4.2 contém uma lista dos principais tipos de memória de semicondutores mais importantes. O mais comum é conhecido como *memória de acesso aleatório* (*random-access memory* — RAM). Contudo, o uso desse termo é um erro, visto que todas as memórias na tabela são de acesso aleatório. Uma característica particular das memórias RAM é possibilitar que novos dados sejam lidos e escritos rapidamente e de modo bastante fácil. Tanto a leitura quanto a escrita são feitas por meio de sinais elétricos.

Outra característica das memórias RAM é o fato de serem voláteis. Uma memória RAM requer um fornecimento de energia constante. Se o fornecimento de energia for interrompido, os dados são perdidos. Portanto, a memória RAM só pode ser utilizada para armazenamento temporário de dados.

A tecnologia das memórias RAM pode ser dividida em estática e dinâmica. Uma *memória RAM dinâmica* é feita de células que armazenam dados com a carga de capacitores. A presença ou a ausência de carga em um capacitor é interpretada como representação do dígito binário 1 ou 0. Como um capacitor tem tendência natural para se descarregar, a RAM dinâmica requer uma regeneração (refresh) de carga periódica para manter os dados armazenados. Na *memória RAM estática*, os valores binários são armazenados usando configurações tradicionais de flip-flops com portas lógicas (veja a descrição de flip-flops no Apêndice A). A memória RAM estática mantém seus dados enquanto houver fornecimento de energia.

Tanto a memória RAM estática quanto a dinâmica são voláteis. A posição de uma memória dinâmica é mais simples e, portanto, menor do que a de uma memória estática. Dessa maneira, uma RAM dinâmica é mais densa (células menores implicam mais células por unidade de área) e mais barata do que uma RAM estática correspondente. Por outro lado, ela requer um circuito de regeneração. No caso de memórias de grande capacidade, o custo fixo do circuito de regeneração é compensado pelo custo menor das células da RAM dinâmica. Portanto, as RAMs dinâmicas tendem a ser mais vantajosas quando a capacidade de memória requerida é maior. Uma observação final é que as RAMs estáticas são, em geral, mais rápidas do que as dinâmicas.

Em contraste com a memória RAM, existe a *memória apenas de leitura* (*read-only memory* — ROM). Como o nome sugere, a memória ROM contém um padrão permanente de dados que não pode ser alterado. Embora seja possível ler uma ROM, não é possível gravar novos dados. Uma aplicação importante de memórias ROM é na microprogramação, discutida na Parte IV. Outras aplicações potenciais são:

- Bibliotecas de sub-rotinas freqüentemente utilizadas.
- Programas de sistema.
- Tabelas de funções.

Se a capacidade de memória requerida for relativamente pequena, a vantagem da utilização de uma memória ROM é que os dados ou programas ficam permanentemente armazenados na memória principal, nunca precisando ser carregados a partir de um dispositivo de armazenamento secundário.

Uma ROM é fabricada como qualquer outra pastilha de circuito integrado, cujos dados são gravados na pastilha durante o processo de fabricação. Isso apresenta dois problemas:

- A etapa de gravação de dados tem um custo fixo relativamente alto, que não depende do número de cópias produzidas.
- Não podem ocorrer erros: se algum bit estiver errado, todo o lote da memória ROM será inutilizado.

Quando apenas um pequeno número de memórias ROM com um dado conteúdo de memória é necessário, uma alternativa mais barata é a *ROM programável* (*programmable ROM* — PROM). Assim como a memória ROM, a PROM é não-volátil e os dados podem ser gravados apenas uma vez. O processo de gravação em uma memória PROM é efetuado eletricamente e pode ser feito pelo fornecedor ou pelo cliente, após a fabricação da pastilha original. Um equipamento especial é necessário para esse processo de gravação ou "programação". As memórias PROM fornecem flexibilidade e conveniência. Essa memória ROM é mais vantajosa no caso de produção em larga escala.

Outra variação da memória apenas de leitura é a memória principalmente de leitura, mais útil em aplicações em que é necessário armazenamento não-volátil e as operações de leitura são muito mais freqüentes do que as de escrita. Há três formas comuns de memória principalmente de leitura: EPROM, EEPROM e memória flash.

A memória EPROM é uma *memória programável apenas de leitura* que pode ser apagada por um processo óptico; os dados podem ser lidos e gravados eletricamente, assim como a PROM. Entretanto, antes de qualquer operação de escrita, todas as células de memória devem ser apagadas, voltando a ter o mesmo valor do estado inicial, pela exposição da pastilha à radiação ultravioleta. Esse processo de apagamento pode ser feito várias vezes; cada ação de apagamento pode levar 20 minutos. A EPROM pode, dessa maneira, ser alterada várias vezes e, como a ROM e a PROM, mantém seus dados quase indefinidamente. Para quantidades equivalentes de área de armazenamento, ela é mais cara que a PROM, contudo tem a vantagem de possibilitar várias atualizações.

Uma forma mais atraente de memória principalmente de leitura é a *memória apenas de leitura programável e apagável eletricamente* (EEPROM). Quaisquer dados podem ser gravados nessa memória sem que seja necessário apagar todo o seu conteúdo anterior; apenas o byte ou os bytes endereçados são atualizados. A operação de escrita de dados nessa memória leva um tempo consideravelmente maior do que a operação de leitura: da ordem de centenas de microssegundos por byte. A EEPROM combina a vantagem de não-volatilidade com a flexibilidade de poder ser atualizada diretamente, por meio das linhas de dados e de endereço usuais e do barramento de controle. Ela é mais cara que a EPROM e também menos densa, fornecendo menor número de bits por pastilha.

Outra forma de memória de semicondutor é a *memória flash* (assim chamada por causa da velocidade com que pode ser programada). Esse tipo de memória, introduzida em meados dos anos 80, apresenta características intermediárias entre a EPROM e a EEPROM, tanto em custo quanto em funcionalidade. Como a EEPROM, a memória flash usa uma tecnologia de apagamento elétrico de dados, podendo ser completamente apagada em poucos segundos, muito mais rápida que a memória EPROM. Além disso, é possível apagar apenas alguns blocos de memória, e não toda a pastilha. Entretanto, essa memória não permite apagar o conteúdo de apenas um byte. Como a EPROM, ela usa apenas um transistor por bit, obtendo assim uma alta densidade (se comparada à EEPROM).

Organização

O elemento básico de uma memória de semicondutor é a célula de memória. Embora possam ser fabricadas usando diferentes tecnologias eletrônicas, todas as células de memória de semicondutor compartilham certas propriedades:

- Exibem dois estados estáveis (ou semi-estáveis), que podem ser utilizados para representar os dígitos binários 1 e 0.
- Um valor pode ser escrito (pelo menos uma vez) em uma célula e o dado gravado define o estado da célula de memória.
- O estado da célula de memória pode ser lido.

A Figura 4.3 mostra a operação de uma célula de memória. A célula geralmente possui três terminais funcionais capazes de carregar um sinal elétrico. O terminal de seleção, como o nome sugere, seleciona uma célula de memória para leitura ou escrita. O terminal de controle indica se a operação é de leitura ou de escrita. Em uma operação de escrita, o terceiro terminal fornece um sinal elétrico que indica se o estado da célula deve ser 1 ou 0. Em uma operação de leitura, o sinal nesse terminal é empregado para indicar o estado da célula. Os detalhes da organização interna, do funcionamento e da temporização de uma célula de memória dependem da tecnologia específica de circuito integrado utilizada, estando além do escopo deste livro. Para nosso propósito, precisamos apenas levar em conta que cada célula pode ser selecionada individualmente para leitura ou escrita.

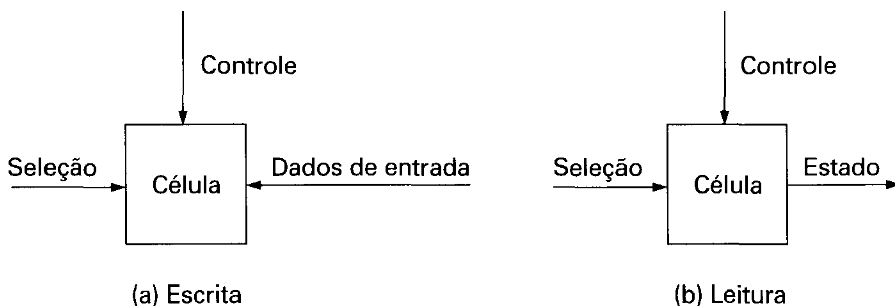


Figura 4.3 Operação de uma célula de memória.

Lógica interna das pastilhas

Assim como nos demais produtos de circuitos integrados, a memória de semicondutor é empacotada em pastilhas (Figura 2.7), cada qual com um grupo de posições de memória.

Vimos anteriormente que, em uma hierarquia de memória, os requisitos de capacidade, velocidade e custo são conflitantes. Esse conflito existe também quando consideramos a organização das células de memória e a lógica funcional de uma pastilha. Um dos aspectos fundamentais do projeto de memórias de semicondutor é o número de bits de dados que podem ser lidos ou escritos simultaneamente. Em um extremo temos uma organização em que o arranjo físico do conjunto de células é igual ao arranjo lógico das palavras na memória (tal como é percebido pelo processador). O conjunto de células é organizado em *P* palavras de B bits cada um. Por exemplo, uma pastilha de 16 Mbits pode ser organizada em 1M palavra de 16 bits. No outro extremo temos a chamada organização de um bit por pastilha, na qual os dados

são lidos ou gravados um bit de cada vez. Apresentamos, a seguir, a organização de uma pastilha de memória DRAM; a organização de uma pastilha de ROM é parecida, embora mais simples.

A Figura 4.4 mostra uma organização típica de uma DRAM de 16 Mbits. Nessa organização, 4 bits são utilizados em cada operação de leitura ou de escrita. A memória é logicamente organizada em quatro matrizes de 2.048 por 2.048 elementos. Vários arranjos físicos são possíveis. Em qualquer caso, os elementos de cada matriz são conectados por linhas de sinais horizontais (linha) e verticais (coluna). Cada linha horizontal é conectada ao terminal de seleção de cada uma das células da linha correspondente na matriz; cada linha vertical é conectada ao terminal de entrada de dados/estado de cada uma das células da coluna correspondente na matriz.

As linhas de endereço fornecem o endereço da palavra a ser selecionada. O número de linhas de endereço necessárias é $\log_2 P$. No nosso exemplo, são necessárias 11 linhas de endereço para selecionar uma das 2.048 linhas horizontais. Essas 11 linhas são conectadas a um decodificador de linha, que tem 11 linhas de entrada e 2.048 linhas de saída. A lógica do decodificador ativa apenas uma das 2.048 saídas, dependendo do padrão de bits nas 11 linhas de entrada ($2^{11} = 2.048$).

Para selecionar uma entre as 2.048 colunas de 4 bits são utilizadas 11 linhas de endereço adicionais. Quatro linhas de dados são usadas para a entrada ou a saída de 4 bits de dados, para uma área de armazenamento temporário de dados. Em uma operação de escrita, o driver de cada bit de dados é ativado, com valor 1 ou 0, de acordo com o sinal de entrada na linha de dados correspondente. Em uma operação de leitura, o valor de cada um dos 4 bits é passado por um amplificador de estado e é exibido na linha de dados correspondente. A linha horizontal seleciona a linha da matriz em que é efetuada a leitura ou a escrita.

Como apenas 4 bits podem ser lidos ou escritos simultaneamente na memória DRAM, várias memórias DRAM devem ser conectadas ao controlador de memória para que uma palavra possa ser lida ou escrita no barramento.

Note que existem apenas 11 linhas de endereço (A0-A10), metade do número que se poderia esperar para uma matriz de 2.048 × 2.048 elementos. Isso é feito para economizar o número de pinos. As 22 linhas de endereço necessárias são passadas usando uma lógica externa à pastilha e multiplexadas sobre as 11 linhas de endereço existentes. Primeiramente, são passados 11 sinais de endereço para a pastilha, utilizados para determinar a linha da matriz; então, os outros 11 sinais de endereço são usados para selecionar a coluna. Esses sinais são acompanhados por um sinal de seleção de endereço de linha (*row address select* — RAS) e de seleção de endereço de coluna (*column address select* — CAS), para controlar a temporização da pastilha.

O uso de linhas de endereço multiplexadas e de matrizes de memória quadradas possibilita aumentar quatro vezes a capacidade da memória a cada nova geração de pastilhas. Cada novo pino de endereço acrescentado possibilita duplicar o número de linhas e colunas da matriz, multiplicando por quatro o tamanho da memória.

A Figura 4.4 inclui também um circuito de regeneração (*refresh*). Toda memória DRAM necessita de uma regeneração periódica dos dados armazenados. Uma técnica de regeneração simples consiste em desabilitar a pastilha de memória DRAM enquanto suas células de dados são regeneradas. Um contador de regeneração é utilizado para percorrer todas as linhas, sendo incrementado cada vez que os dados de uma linha da matriz são regenerados. Para cada linha, os sinais de saída do contador são passados para o decodificador de linha e a linha de sinal RAS é ativada. Isso faz com que cada célula da linha seja regenerada.

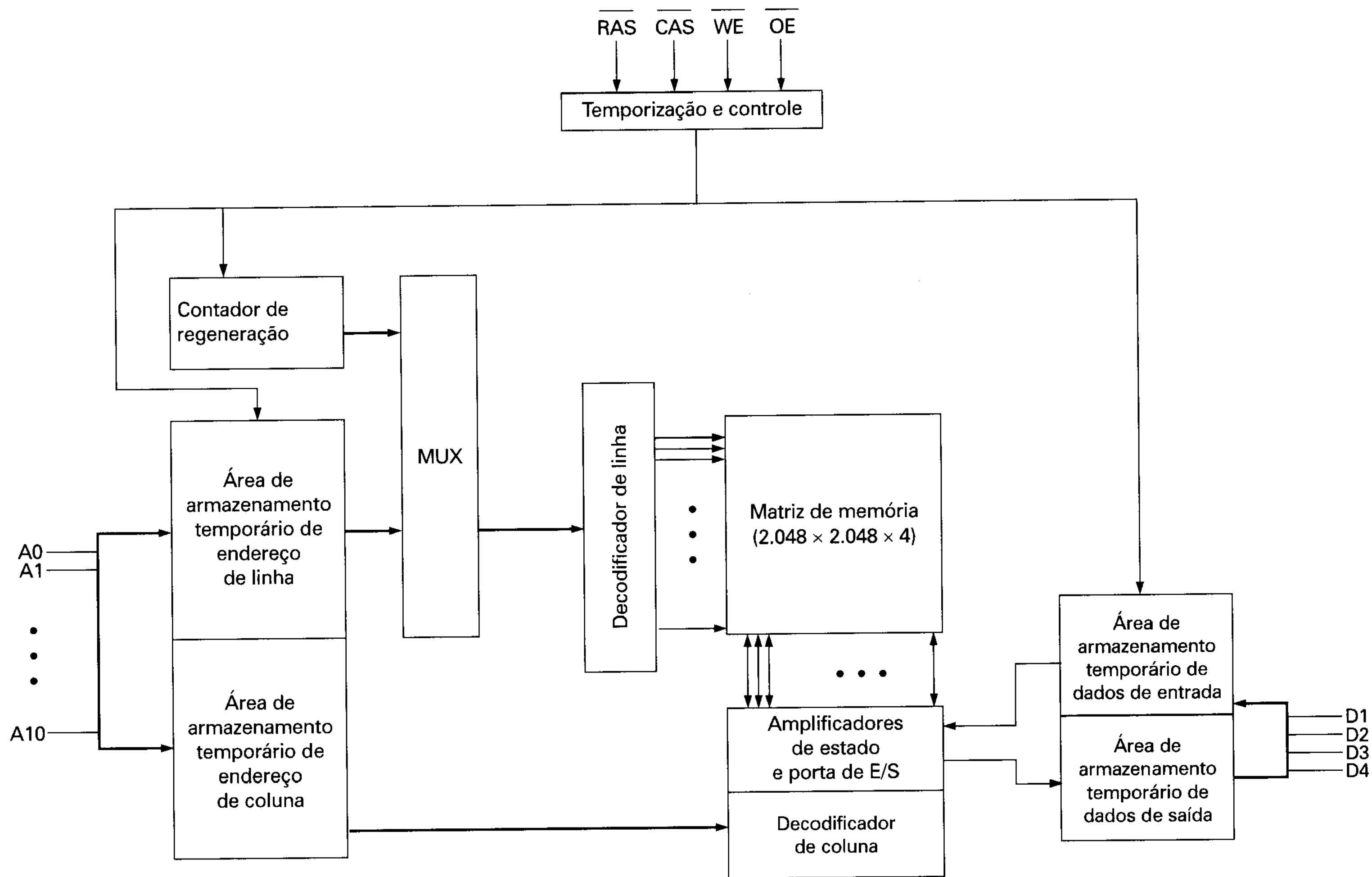
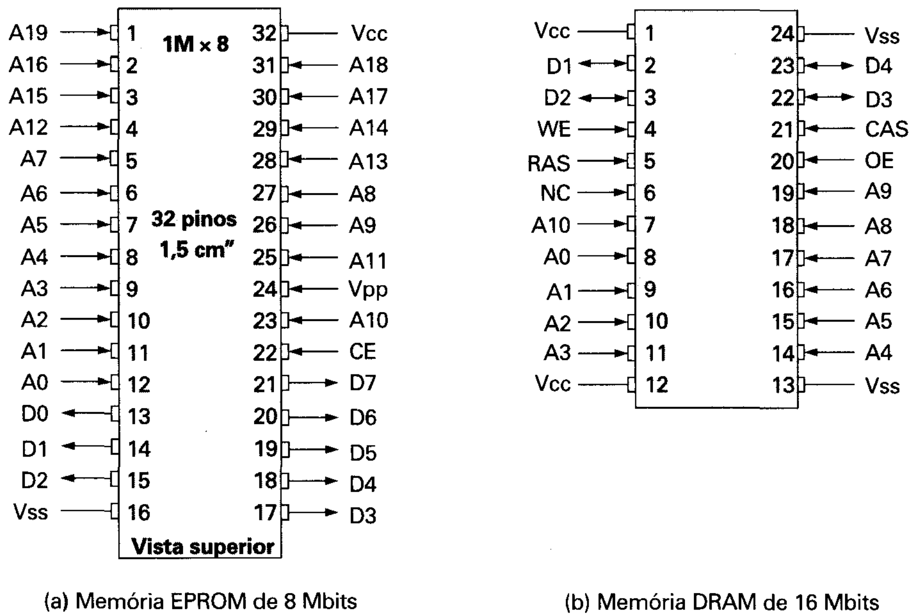


Figura 4.4 DRAM típica de 16 Mbits (4M × 4).



(a) Memória EPROM de 8 Mbits (b) Memória DRAM de 16 Mbits

Figura 4.5 Sinais e pinos típicos de empacotamentos de memória.

Empacotamento das pastilhas

Como vimos no Capítulo 2, um circuito integrado é empacotado dentro de uma cápsula que contém pinos para sua conexão com circuitos externos.

A Figura 4.5a mostra um exemplo de empacotamento de uma memória EPROM de 8 Mbits, organizada como uma matriz de 1M × 8. Nesse caso, a organização adotada é de um empacotamento com uma palavra por pastilha. A cápsula inclui 32 pinos, constituindo um dos tamanhos padrão de pastilhas de memória. Os pinos contêm as seguintes linhas de sinal:

- O endereço da palavra que está sendo acessada. Para 1M palavra, um total de 20 pinos (A0 - A19) são necessários (2^{20} = 1M).
- Os dados a serem lidos são constituídos de oito linhas (D0-D7).
- A energia fornecida à pastilha (Vcc).
- Um pino de terra (Vss).
- Um pino de habilitação da pastilha (*chip enable* — CE). Existindo mais de uma pastilha de memória, como elas estão conectadas ao mesmo barramento de endereços, o pino CE é usado para indicar se o endereço é válido ou não para a]pastilha. Ele é ativado por um circuito lógico conectado aos bits mais significativos do barramento de endereços (por exemplo, pelos bits de endereço acima de A19). O uso desse sinal é ilustrado a seguir.
- Um pino de voltagem para programação (Vpp) que é fornecido durante a programação da pastilha (operações de escrita).

Uma configuração de pinos típica de memórias DRAM é mostrada na Figura 4.5b, para uma pastilha de 16 Mbits, organizada como uma matriz de 4M × 4. Há várias diferenças em relação à pastilha de ROM. Como a memória RAM pode ser atualizada, os pinos de dados são de entrada e saída. Pinos de habilitação de escrita (*write enable* — WE) e habilitação de saída

(*output enable* — OE) são usados para indicar se a operação é de escrita ou de leitura. Como o acesso à memória DRAM é feito em termos de linhas e colunas e as linhas de endereço são multiplexadas, são necessários apenas 11 pinos de endereço para especificar 4M combinações de linhas e colunas ($2^{11} \times 2^{11} = 2^{22} = 4M$). As funções dos pinos de seleção de endereço de linha (RAS) e de seleção de endereço de coluna (CAS) foram discutidas anteriormente.

Organização em módulos

Se uma pastilha de RAM contém apenas 1 bit por palavra, é claro que o número de pastilhas necessárias é igual ao número de bits por palavra. Por exemplo, a Figura 4.6 mostra como um módulo de memória de 256K palavras de 8 bits pode ser organizado. Para 256K palavras, é necessário um endereço de 18 bits, que é fornecido ao módulo por alguma fonte externa (por exemplo, pelas linhas de endereço do barramento ao qual o módulo está conectado). O endereço é apresentado às oito pastilhas de 256K × 1 bit, cada um dos quais fornecendo a entrada/saída de 1 bit.

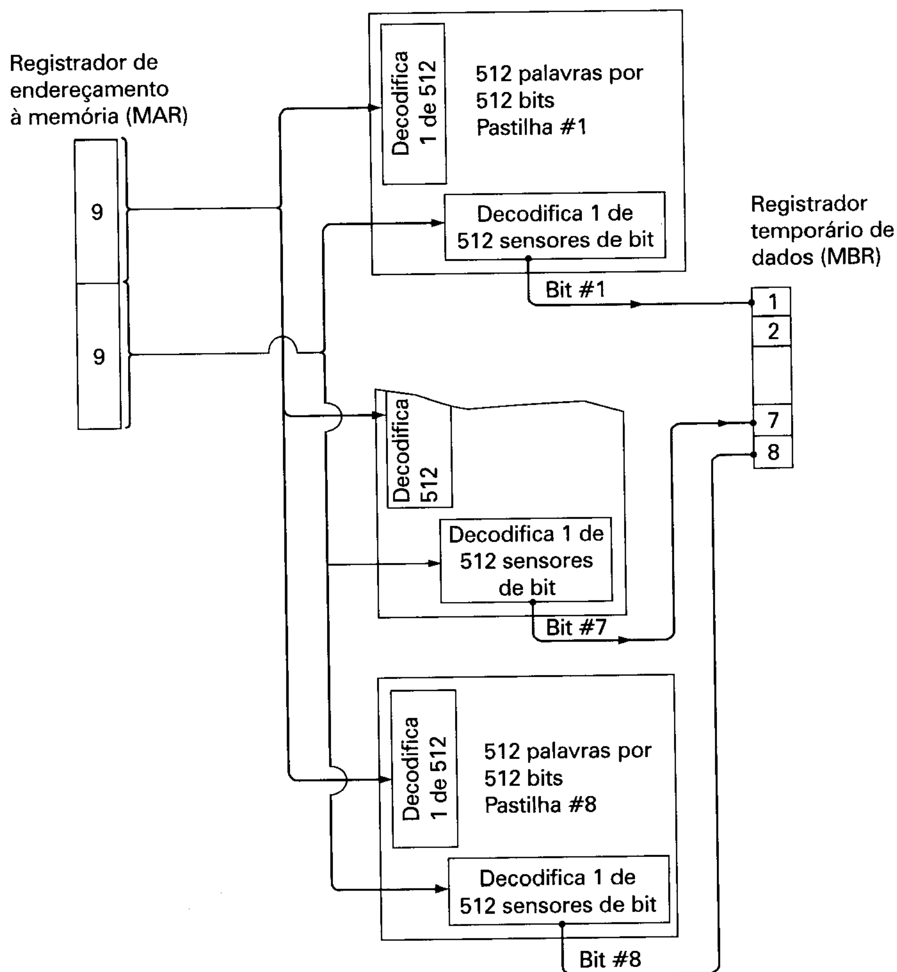


Figura 4.6 Organização de memória de 256 Kbytes.

Esse arranjo funciona desde que o tamanho da memória seja igual ao número de bits por pastilha. Caso seja necessária uma memória de maior capacidade, é preciso agrupar um conjunto de pastilhas. A Figura 4.7 mostra uma organização possível para uma memória de 1M palavra de 8 bits. Nesse caso, temos quatro colunas de pastilhas, cada qual com 256K palavras, arranjadas como na Figura 4.6. Para 1M palavra, são necessárias 20 linhas de endereço. Os 18 bits menos significativos são enviados a cada um dos 32 módulos. Os 2 bits mais significativos são introduzidos em um módulo de lógica de seleção de grupo, que envia um sinal de habilitação de pastilha para uma das quatro colunas de módulos.

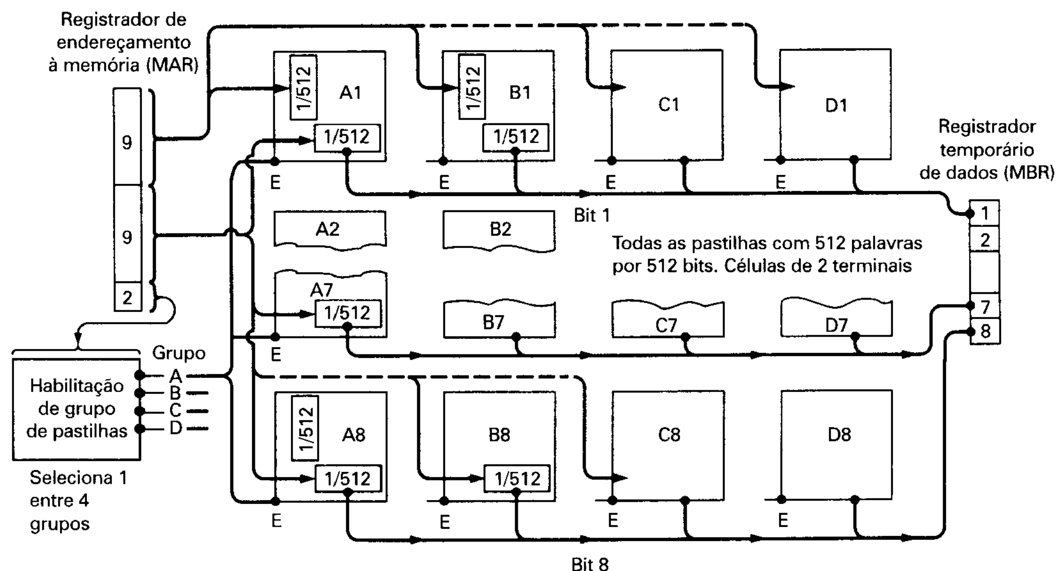


Figura 4.7 Organização de memória de 1 Mbyte.

Correção de erros

Todo sistema de memória de semicondutor está sujeito a erros. Esses erros podem ser classificados como falhas graves ou erros moderados. Uma falha grave constitui um defeito físico permanente; a célula ou células de memória afetadas não são capazes de armazenar os dados com segurança, podendo permanecer sempre com valor 0 ou 1 ou variar entre 0 e 1 aleatoriamente. Falhas graves podem ser causadas pelo uso excessivo em ambiente inadequado, por defeitos de fabricação ou por desgaste. Um erro moderado é um evento aleatório e não-destrutivo, que altera o conteúdo de uma ou mais posições de memória sem danificar a memória. Erros moderados podem ser causados por problemas de fornecimento de energia ou pela presença de partículas alfa. Essas partículas resultam de decaimento radiativo e são lamentavelmente comuns, pois pequenas quantidades de núcleos radiativos são encontradas em quase todos os materiais. Tanto falhas graves quanto erros moderados são obviamente indesejáveis; a maioria dos sistemas de memória principal modernos inclui uma lógica de detecção e correção de erros.

A Figura 4.8 ilustra, em termos gerais, o processo de detecção e correção de erros. Quando um dado é armazenado na memória, é feito um cálculo envolvendo esse dado, representado na Figura 4.8 pela função *f*, para produção de um código. Esse código é armazenado

juntamente com os dados. Assim, se uma palavra de M bits for armazenada e o código tiver K bits, o tamanho verdadeiro da palavra armazenada será $M + K$ bits.

Quando a palavra armazenada anteriormente é lida, o código é utilizado para detectar e, possivelmente, corrigir erros. Um novo conjunto de K bits de código é gerado a partir dos M bits de dados lidos e comparado com o código armazenado. A comparação pode produzir um dos três resultados a seguir:

- Nenhum erro é detectado. Nesse caso, os bits de dados obtidos são enviados.
- Um erro é detectado e é possível corrigi-lo. Nesse caso, os bits de dados e os bits de correção de erros são introduzidos em um circuito de correção de erro, que produz um conjunto corrigido de M bits para serem enviados.
- Um erro é detectado, mas não é possível corrigi-lo. Uma condição de erro associado ao evento é relatada.

Os códigos que operam desse modo são denominados *códigos de correção de erros*. Um código é caracterizado pelo número de bits incorretos que ele é capaz de detectar e corrigir em uma palavra.

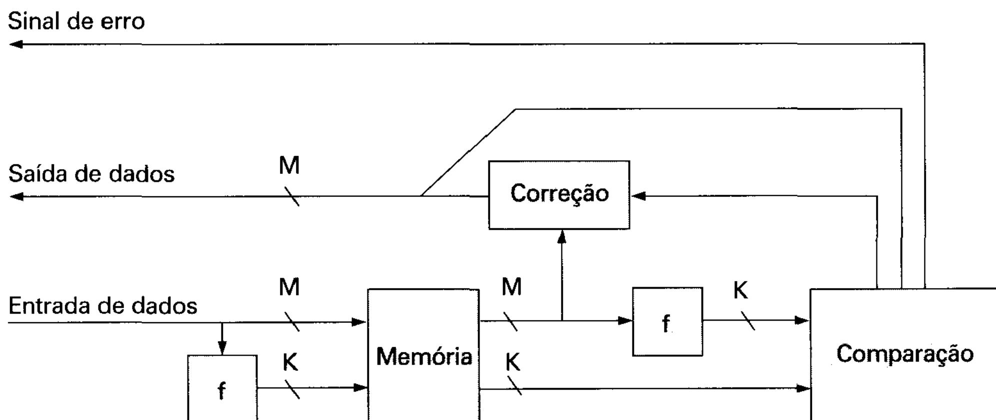


Figura 4.8 Código de correção de erros.

O código de correção de erros mais simples é o *código de Hamming*, projetado por Richard Hamming na Bell Laboratories. A Figura 4.9 utiliza diagramas de Venn para ilustrar o uso desse código para palavras de 4 bits ($M = 4$). Os três círculos que se interceptam definem sete compartimentos. Os quatro bits de dados são atribuídos a compartimentos internos (Figura 4.9a), os compartimentos restantes são preenchidos com os chamados bits de paridade. Cada *bit de paridade* é escolhido de modo que o número total de 1s em seu círculo seja par (Figura 4.9b). Dessa maneira, como o círculo A inclui três valores 1s, o *bit de paridade* nesse círculo é 1. Caso um erro modifique um dos bits de dados (Figura 4.9c), o bit alterado pode ser facilmente encontrado. Verificando-se os bits de paridade, encontramos divergências nos círculos A e C, mas não no círculo B. Apenas um dos sete compartimentos pertence aos círculos A e C, mas não ao B. Portanto, o erro pode ser corrigido alterando-se o bit desse compartimento.

Para tornar os conceitos envolvidos mais claros, projetamos um código para detectar e corrigir erros que ocorram em um único bit, usando palavras de 8 bits.

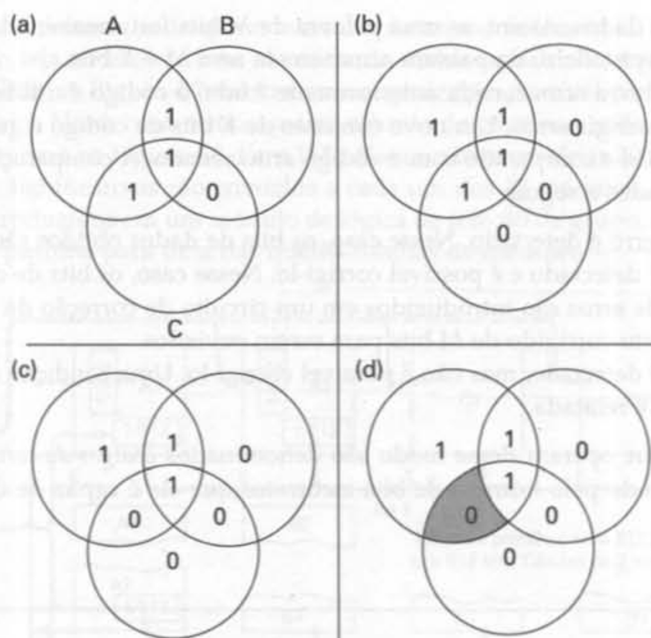


Figura 4.9 Código de correção de erros de Hamming.

Primeiramente determinamos o tamanho que o código deve ter. De acordo com a Figura 4.8, a lógica de comparação tem como entrada dois valores de *K* bits. A comparação é feita bit a bit, aplicando a operação lógica ou-exclusivo sobre as duas entradas: cada bit da palavra resultante dessa operação, denominado *palavra síndrome*, tem valor 0 ou 1, conforme os valores dos bits na mesma posição das duas entradas sejam iguais ou diferentes.

A palavra síndrome tem, portanto, *K* bits de tamanho e armazena valores entre 0 e $2^K - 1$. O valor 0 indica que nenhum erro foi detectado, e os $2^K - 1$ valores restantes podem ser utilizados para indicar qual é o bit errado, caso haja um erro. Como pode ocorrer um erro em qualquer um dos *M* bits de dados ou dos *K* bits de teste, devemos ter:

$$2^K - 1 \geq M + K$$

Essa equação fornece o número de bits de código necessários para corrigir erros em um único bit, usando uma palavra com *M* bits de dados. A Tabela 4.3 apresenta uma lista do número de bits de teste necessários para palavras de vários tamanhos.

Tabela 4.3 Aumento no tamanho de uma palavra com a correção de erros

	Correção de erro único		Correção de erro único/detecção de erro duplo	
Bits de dados	**Bits de teste**	**Porcentagem de aumento**	**Bits de teste**	**Porcentagem de aumento**
8	4	50	5	62,5
16	5	31,25	6	37,5
32	6	18,75	7	21,875
64	7	10,94	8	12,5
128	8	6,25	9	7,03
256	9	3,52	10	3,91

A partir desta tabela, podemos ver que uma palavra de 8 bits de dados necessita de 4 bits de teste. Por conveniência, gostaríamos de ter uma palavra síndrome de 4 bits com as seguintes características:

- Se todos os bits da palavra síndrome têm valor 0s, não ocorreu nenhum erro.
- Se a palavra síndrome contém apenas um bit com valor 1, ocorreu erro em um dos 4 bits de teste. Nenhuma correção é necessária.
- Se a palavra síndrome contém mais de um bit com valor 1, o valor numérico da palavra síndrome indica a posição do bit em que ocorreu erro. A palavra é corrigida invertendo-se o valor desse bit do dado.

Para satisfazer essas características, os bits de dados e de teste são organizados em uma palavra de 12 bits, conforme representado na Figura 4.10. As posições dos bits são numeradas de 1 a 12. As posições de bits cujo número é uma potência de 2 são reservadas como bits de teste. Os bits de teste são calculados como se mostra a seguir, onde o símbolo representa a operação ou-exclusivo:

Posição do bit	Número da posição	Bit de teste	Bit de dados
12	1 1 0 0		M8
11	1 0 1 1		M7
10	1 0 1 0		M6
9	1 0 0 1		M5
8	1 0 0 0	C8	
7	0 1 1 1		M4
6	0 1 1 0		M3
5	0 1 0 1		M2
4	0 1 0 0	C4	
3	0 0 1 1		M1
2	0 0 1 0	C2	
1	0 0 0 1	C1	

Figura 4.10 Arranjo de bits de dados e bits de teste.

$$
\begin{array}{llllllll}
C1 = M1 \oplus M2 \oplus & & M4 \oplus & M5 \oplus & & M7 & \\
C2 = M1 \oplus & M3 \oplus & M4 \oplus & & M6 \oplus & M7 & \\
C4 = \quad M2 \oplus & M3 \oplus & M4 \oplus & & & & M8 \\
C8 = & & & M5 \oplus & M6 \oplus & M7 \oplus & M8
\end{array}
$$

Cada bit de teste opera sobre todas as posições de bits de dados cujo número contém um valor 1 na coluna de posição correspondente. Assim, as posições 3, 5, 7, 9 e 11 de bits de dados contêm, todas elas, o termo 2^0; as posições de bit 3, 6, 7, 10 e 11 contêm o termo 2^1; as posições de bit 5, 6, 7 e 12 contêm o termo 2^2; e as posições de bit 9, 10, 11 e 12 contêm o termo 2^3. Em outras palavras, a posição de bit n é testada pelos bits C_i, tal que $\sum i = n$. Por exemplo, a posição 7 é testada pelos bits nas posições 4, 2 e 1, pois 7 = 4 + 2 + 1.

Verificamos que esse esquema funciona com um exemplo. Suponha que a palavra de 8 bits dada como entrada é 00111001, com bit de dados M1 na posição mais à direita. Os cálculos feitos são mostrados a seguir:

$$
\begin{aligned}
C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C2 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C4 &= 0 \oplus 0 \oplus 1 \oplus 0 = 1 \\
C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

Suponha agora que o terceiro bit de dados (M3) foi alterado de 0 para 1, contendo um erro. Quando os bits de teste são recalculados, obtemos:

$$
\begin{aligned}
C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C2 &= 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0 \\
C4 &= 0 \oplus 1 \oplus 1 \oplus 0 = 0 \\
C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

Quando os novos bits de teste são comparados com os anteriores, a seguinte palavra síndrome é formada:

$$
\begin{array}{rcccc}
 & C8 & C4 & C2 & C1 \\
 & 0 & 1 & 1 & 1 \\
\oplus & 0 & 0 & 0 & 1 \\
\hline
 & 0 & 1 & 1 & 0
\end{array}
$$

O resultado é 0110, o que indica que o conteúdo do bit de posição 6, que corresponde ao terceiro bit de dados (M3), está errado.

A Figura 4.11 mostra o cálculo anterior. Os bits de dados e de teste são posicionados adequadamente em uma palavra de 12 bits. Arranjando os números de posição de cada bit de dados em colunas, os 1s em cada linha indicam os bits de dados testados pelo bit de teste daquela linha. Como o resultado é afetado apenas pelos bits com valor 1, apenas as colunas que contêm 1s são marcadas para identificação. Os bits de teste podem então ser calculados ao longo das linhas. Os resultados são mostrados para os bits de dados originais e para os bits de dados com erro.

O código descrito anteriormente é conhecido como um código de correção de erro único (*single-error-correcting* — SEC). Usualmente, uma memória de semicondutor é equipada com um código de detecção de erro duplo e correção de erro único (*single-error-correcting, double-error-detecting* — SEC-DED). Como mostra a Tabela 4.3, esses códigos requerem um bit a mais do que os códigos SEC.

Posição de bit	12	11	10	9	8	7	6	5	4	3	2	1	
Bit de dados	M8	M7	M6	M5		M4	M3	M2		M1			
Bit de teste					C8				C4		C2	C1	
	1	1	1	1		0	0	0		0			C8 0
	1	0	0	0		1	1	0		0			C4 1
	0	1	1	0		1	1	0		1			C2 1
	0	1	0	1		1	0	1		1			C1 1
Palavra armazenada como:	0	0	1	1	0	1	0	0	1	1	1	1	
Palavra lida como:	0	0	1	1	0	1	1	0	1	1	1	1	
	1	1	1	1		0	0	0		0			C8 0
	1	0	0	0		1	1	1		0			C4 0
	0	1	1	0		1	1	1		1			C2 0
	0	1	0	1		1	0	1		1			C1 1

Figura 4.11 Degeneração de bit de teste.

A Figura 4.12 ilustra como funciona um código SEC-DEC para uma palavra de dados de 4 bits. A seqüência mostra que, se ocorrerem dois erros (Figura 4.12c), o procedimento de teste segue para o passo (d), piorando o problema e criando um terceiro erro (e). Para evitar esse problema, é acrescentado um oitavo bit, cujo valor é calculado de modo que o número total de 1s no diagrama seja par. Esse bit de paridade extra detecta o erro (f).

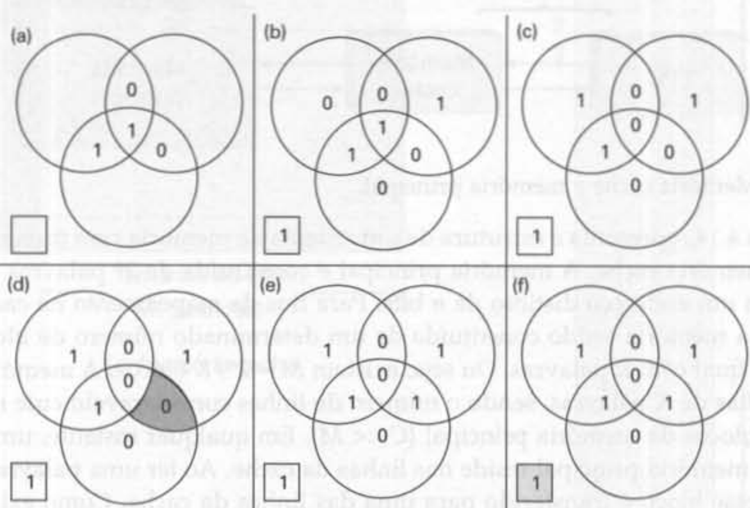


Figura 4.12 Código de Hamming SEC-DED.

O uso de um código de correção de erros aumenta a confiabilidade da memória, ao custo de um acréscimo de complexidade. Em uma organização de memória de um bit por pastilha, o código SEC-DED é geralmente considerado adequado. Por exemplo, as implementações da linha 30xx da IBM utilizam um código SEC-DED de 8 bits para cada 64 bits de dados da memória principal. Dessa maneira, o tamanho da memória principal é, na verdade, cerca de 12% maior do que aparenta para o usuário. Os computadores VAX usam um código SEC-DED de 7 bits para cada 32 bits de memória, com um acréscimo de 22% no tamanho da memória. Diversas memórias DRAM modernas utilizam 9 bits de teste para cada 128 bits de dados, com um acréscimo de 7% no tamanho da memória (Sharma, 1997).

4.3 MEMÓRIA CACHE

Princípios fundamentais

O uso de memória cache visa obter uma velocidade de acesso à memória próxima da velocidade das memórias mais rápidas e, ao mesmo tempo, disponibilizar no sistema uma memória de grande capacidade, a um custo equivalente ao das memórias de semicondutor mais baratas. O conceito de memória cache é mostrado na Figura 4.13. Uma memória principal relativamente grande e lenta é combinada com uma memória cache menor e mais rápida. A memória cache contém uma cópia de partes da memória principal. Quando o processador deseja ler uma palavra da memória, é realizado um teste para determinar se a palavra está na memória cache. Se estiver, ela é fornecida ao processador. Caso contrário, um bloco de dados da memória principal, constituído de um número fixo de palavras, é lido para a memória cache e em seguida a palavra requerida é entregue ao processador. Em razão do fenômeno de localidade de referências, quando um bloco de dados é trazido para a memória cache para satisfazer uma dada referência à memória, é provável que futuras referências sejam feitas para outras palavras desse bloco.

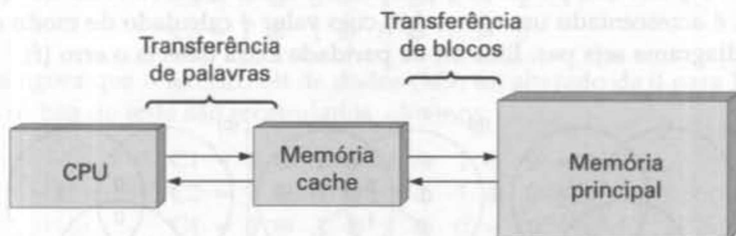


Figura 4.13 Memória cache e memória principal.

A Figura 4.14 representa a estrutura de um sistema de memória com uma memória principal e uma memória cache. A memória principal é constituída de 2^n palavras endereçáveis, cada qual com um endereço distinto de n bits. Para fins de mapeamento na cache, podemos considerar essa memória sendo constituída de um determinado número de blocos de tamanho fixo, cada qual com K palavras. Ou seja, existem $M = 2^n / K$ blocos. A memória cache consiste em C **linhas** de K palavras, sendo o número de linhas consideravelmente menor do que o número de blocos da memória principal ($C << M$). Em qualquer instante, um subconjunto dos blocos da memória principal reside nas linhas da cache. Ao ler uma palavra de um bloco de memória, esse bloco é transferido para uma das linhas da cache. Como existe um maior número de blocos do que linhas da cache, não é possível que uma linha armazene um mesmo bloco permanentemente. Assim, cada linha inclui um rótulo que identifica qual é o bloco de memória nela armazenado. Esse rótulo é geralmente uma parte do endereço de memória principal, como no decorrer desta seção.

A Figura 4.15 mostra a operação de leitura. O processador gera um endereço, RA, da palavra a ser lida. Se essa palavra estiver contida na memória cache, ela será entregue ao processador. Caso contrário, o bloco que contém essa palavra será carregado na memória cache e a palavra será entregue ao processador. A Figura 4.15 mostra que essas duas últimas operações ocorrem paralelamente, refletindo a organização apresentada na Figura 4.16, típica de memórias cache modernas. Nessa organização, a memória cache é conectada ao processador

por meio de linhas de dados, de controle e de endereço. As linhas de dados e de endereço são também conectadas a áreas de armazenamento temporário de dados e de endereço, que se conectam ao barramento do sistema, por meio do qual é feito o acesso à memória principal. Quando uma palavra requerida estiver contida na memória cache (acesso com acerto na cache — cache hit), as áreas de armazenamento temporário de dados e de endereço são desabilitadas e a comunicação ocorre apenas entre o processador e a memória cache, sem ocasionar tráfego no barramento do sistema. Caso contrário, isto é, se a palavra requerida não estiver contida na memória cache (acesso com falha na cache — cache miss), o endereço desejado é carregado no barramento do sistema e os dados requeridos serão transferidos por meio da área de armazenamento temporário de dados, tanto para a cache quanto para a memória principal. Em outras organizações, a memória cache é fisicamente interposta entre o processador e a memória principal para quaisquer comunicações nas linhas de dados, de endereço e de controle. Nesse último caso, quando não é encontrada uma palavra na memória cache, isto é, quando ocorre uma falha na cache, a palavra desejada é primeiramente lida na memória cache e então transferida dessa memória para o processador.

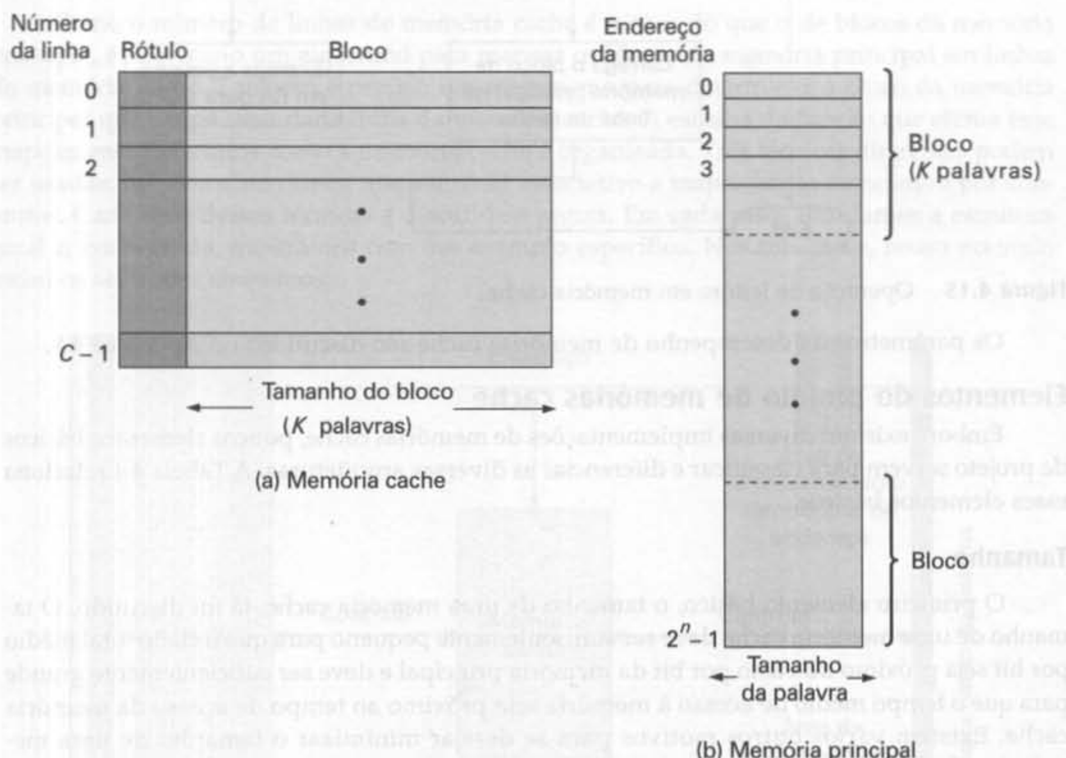


Figura 4.14 Estrutura da memória cache e da memória principal.

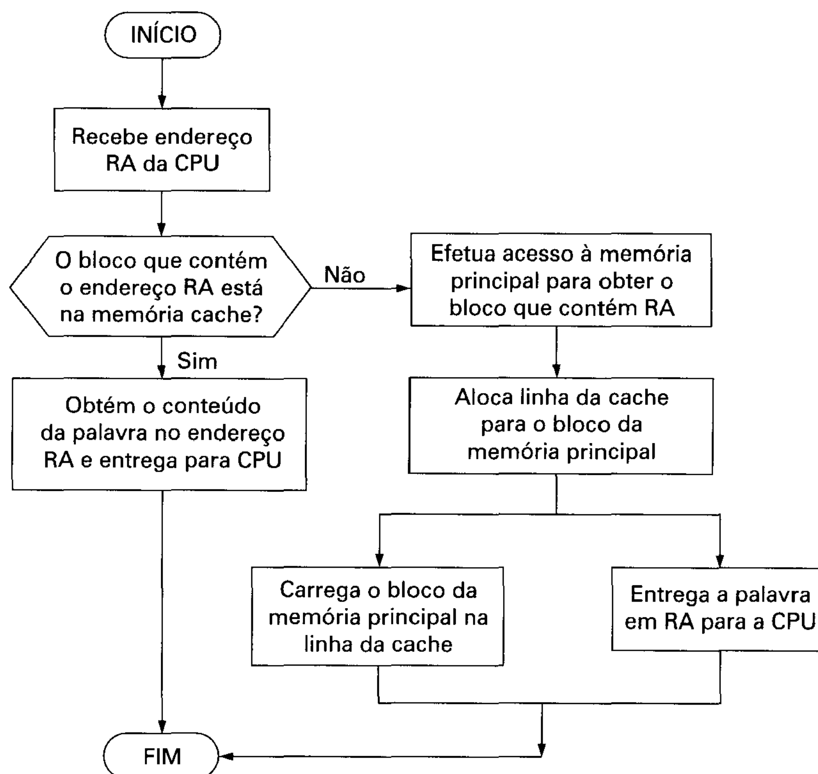


Figura 4.15 Operação de leitura em memória cache.

Os parâmetros de desempenho de memórias cache são discutidos no Apêndice 4A.

Elementos do projeto de memórias cache

Embora existam diversas implementações de memórias cache, poucos elementos básicos de projeto servem para classificar e diferenciar as diversas arquiteturas. A Tabela 4.4 relaciona esses elementos básicos.

Tamanho

O primeiro elemento básico, o tamanho de uma memória cache, já foi discutido. O tamanho de uma memória cache deve ser suficientemente pequeno para que o custo total médio por bit seja próximo do custo por bit da memória principal e deve ser suficientemente grande para que o tempo médio de acesso à memória seja próximo ao tempo de acesso da memória cache. Existem vários outros motivos para se desejar minimizar o tamanho de uma memória cache. Quanto maior é a memória cache, maior é o número de portas envolvidas no seu endereçamento. Como resultado, memórias cache grandes tendem a ser ligeiramente mais lentas do que as pequenas — mesmo quando construídas com a mesma tecnologia de circuito integrado e colocadas no mesmo lugar na pastilha e na placa de circuito. O tamanho de uma memória cache também é limitado pelo tamanho da pastilha e da placa de circuito. O Apêndice 4A mostra que diversos estudos realizados sugerem que memórias cache com tamanho entre 1K e 512K palavras são mais eficazes. Como o desempenho

de uma memória cache é muito sensível à natureza da carga de trabalho imposta, é impossível determinar um tamanho *ótimo*.

Tabela 4.4 Elementos de projeto de memórias cache

Tamanho da memória cache	**Política de escrita**
Função de mapeamento	Escrita direta (write-through)
Direto	Escrita de volta (write-back)
Associativo	Escrita única (write-once)
Associativo por conjuntos	**Tamanho da linha**
Algoritmo de substituição	**Número de memórias cache**
Menos recentemente usado (LRU)	Um ou dois níveis
O primeiro a chegar é o primeiro a sair (FIFO)	Unificada ou separada
Menos freqüentemente usado (LFU)	
Aleatório	

Função de mapeamento

Como o número de linhas de memória cache é menor do que o de blocos da memória principal, é necessário um algoritmo para mapear os blocos da memória principal em linhas da memória cache. Também é preciso um mecanismo para determinar o bloco da memória principal que ocupa uma dada linha da memória cache. A escolha da função que efetua esse mapeamento determina como a memória cache é organizada. Três técnicas diferentes podem ser usadas: mapeamento direto, mapeamento associativo e mapeamento associativo por conjuntos. Cada uma dessas técnicas é discutida a seguir. Em cada caso, abordamos a estrutura geral e, em seguida, mostramos com um exemplo específico. Nos três casos, nosso exemplo inclui os seguintes elementos:

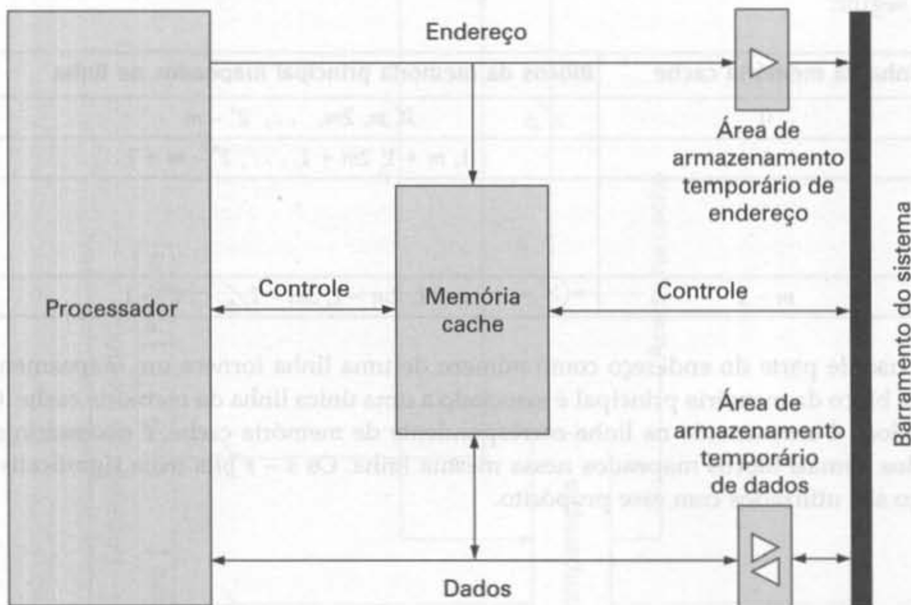


Figura 4.16 Organização típica de uma memória cache.

- A memória cache pode conter 64 Kbytes.
- Os dados são transferidos entre a memória principal e a memória cache em blocos de 4 bytes. Isso quer dizer que a memória cache é organizada com $16K = 2^{14}$ linhas de 4 bytes cada uma.
- A memória principal com 16 Mbytes, sendo cada byte endereçável diretamente por um endereço de 24 bits ($2^{24} = 16M$). Desse modo, para fins de mapeamento, a memória principal é vista como constituída de 4M blocos de 4 bytes cada um.

Na técnica mais simples, conhecida como **mapeamento direto**, cada bloco da memória principal é mapeado em uma única linha de cache. A Figura 4.17 mostra o mecanismo geral. O mapeamento é expresso pela equação:

$$i = j \text{ módulo } m$$

onde:

i = número da linha da memória cache
j = número do bloco da memória principal
m = número de linhas na memória cache

A função de mapeamento é implementada facilmente usando o endereço. Nos acessos à memória cache, um endereço da memória principal pode ser visto como constituído de três campos. Os w bits menos significativos identificam uma única palavra ou byte dentro de um bloco da memória principal; nas máquinas mais modernas, um endereço representa um endereço ao nível de byte. Os s bits restantes especificam um dos 2^s blocos da memória principal. Eles são interpretados na memória cache como compostos de um rótulo de $s - r$ bits (mais significativos) e de um número de linha de r bits que identifica uma das $m = 2^r$ linhas da memória cache. Cada bloco da memória principal é mapeado em uma linha da memória cache como a seguir:

Linha da memória cache	**Blocos da memória principal mapeados na linha**
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m + 1, 2m + 1, \ldots, 2^s - m + 1$
. . .	. . .
$m - 1$	$m - 1, 2m - 1, 3m - 1, \ldots, 2^s - 1$

O uso de parte do endereço como número de uma linha fornece um mapeamento em que cada bloco da memória principal é associado a uma única linha da memória cache. Quando um bloco é armazenado na linha correspondente de memória cache, é necessário distingui-lo dos demais blocos mapeados nessa mesma linha. Os $s - r$ bits mais significativos do endereço são utilizados com esse propósito.

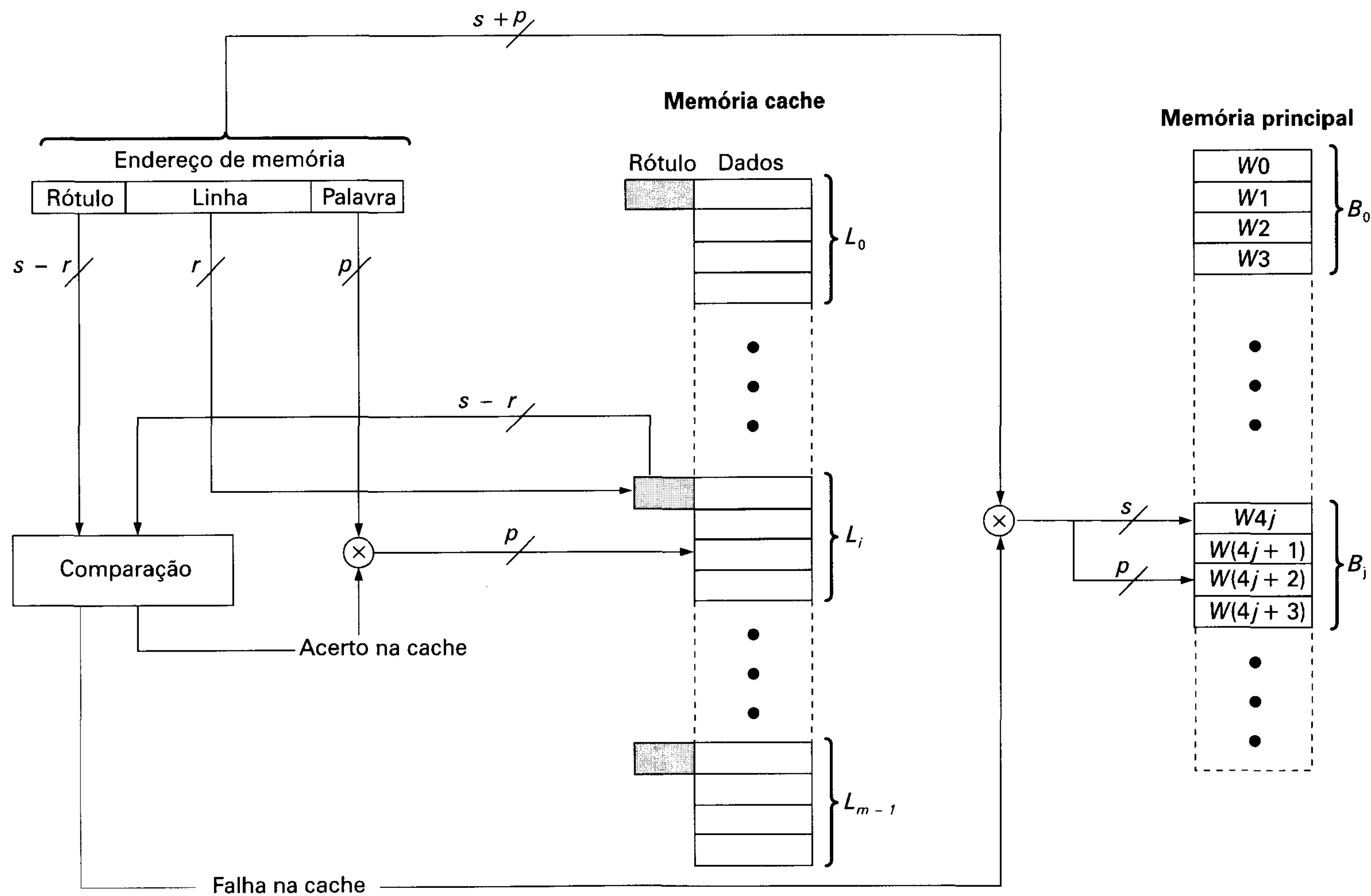


Figura 4.17 Organização de memória cache com mapeamento direto (Hwang, 1993).

A Figura 4.18 mostra o uso de mapeamento direto para nosso sistema exemplo[2]. Nesse exemplo, $m = 16K = 2^{14}$ e $i = j$ módulo 2^{14}. O mapeamento é mostrado a seguir:

Linha da memória cache	**Endereço de início do bloco de memória**
0	000000, 010000, . . . , FF0000
1	000004, 010004, . . . , FF0004
. . .	. . .
$m - 1$	00FFFC, 01FFFC, . . . , FFFFFC

Note que os rótulos de quaisquer dos dois blocos mapeados em uma mesma linha de memória cache são distintos. Assim, os blocos 000000, 010000, . . . , FF0000 têm rótulos de números 00, 01, . . . , FF, respectivamente.

Retornando à Figura 4.15, vemos que a operação de leitura funciona da maneira a seguir. A memória cache recebe um endereço de 24 bits. Um número de linha de 14 bits é utilizado como índice para acessar uma determinada linha de memória cache. Caso o número contido no campo de rótulo de 8 bits seja igual ao número do rótulo armazenado nessa linha, então o número no campo de palavra de 2 bits será utilizado para selecionar um dos 4 bytes da linha. Caso contrário, os 22 bits que compõem os campos de linha e de rótulo serão utilizados para obter o bloco na memória principal. O endereço realmente usado para obter o bloco é composto por esses 22 bits seguidos de dois bits de valor 0. São trazidos os quatro bytes localizados a partir desse endereço de início de bloco.

A técnica de mapeamento direto é simples e tem custo de implementação baixo. Sua principal desvantagem é que cada bloco é mapeado em uma posição fixa na memória cache. Dessa maneira, se um programa fizer repetidas referências a palavras de dois blocos distintos, mapeados em uma mesma linha, esses blocos serão trocados continuamente na memória cache e a taxa de acertos à memória cache será baixa.

O **mapeamento associativo** evita as desvantagens do mapeamento direto, permitindo que cada bloco da memória principal seja carregado em qualquer linha de memória cache. Nesse caso, a lógica de controle da memória cache interpreta um endereço de memória como constituído simplesmente de um rótulo e um campo de palavra. O rótulo identifica um bloco da memória principal de modo unívoco. Para determinar se um bloco está na memória cache, a lógica de controle da memória cache deve comparar simultaneamente o campo de rótulo do endereço do bloco acessado com os rótulos de todas as linhas. A Figura 4.19 mostra essa lógica.

2 Nesta figura, assim como nas subseqüentes, os valores de endereço e de dados são representados na notação hexadecimal, que é descrita no apêndice do Capítulo 8.

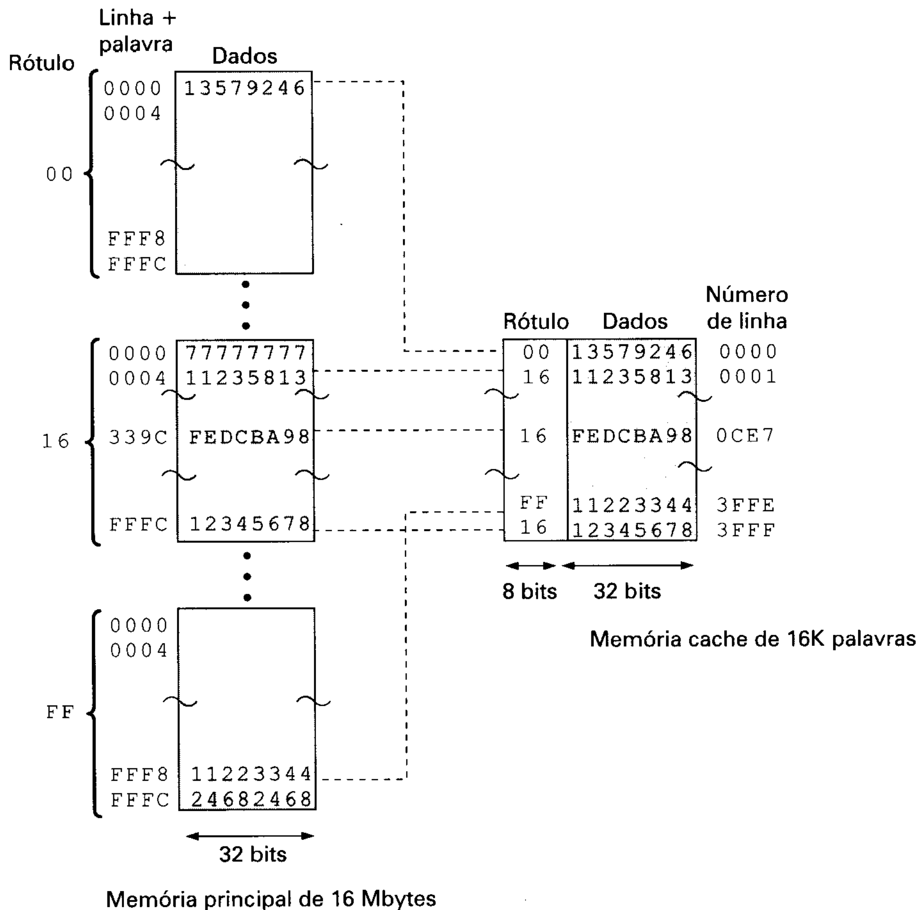


Figura 4.18 Exemplo de mapeamento direto.

A Figura 4.20 apresenta o uso do mapeamento associativo no nosso exemplo. Um endereço de memória principal é composto de um rótulo de 22 bits e de um número de byte de 2 bits. Os 22 bits do rótulo são armazenados junto com os 32 bits de dados do bloco em cada linha de memória cache. Note que o rótulo corresponde aos 22 bits mais à esquerda (mais significativos) do endereço. Dessa maneira, o endereço hexadecimal de 24 bits 16339C tem rótulo igual a 058CE7. Isso pode ser visto mais facilmente utilizando a notação binária:

endereço de memória	0001	0110	0011	0011	1001	1100	(binário)
	1	6	3	3	9	C	(hex.)
rótulo (22 bits mais à esquerda)	00	0101	1000	1100	1110	0111	(binário)
	0	5	8	C	E	7	(hex.)

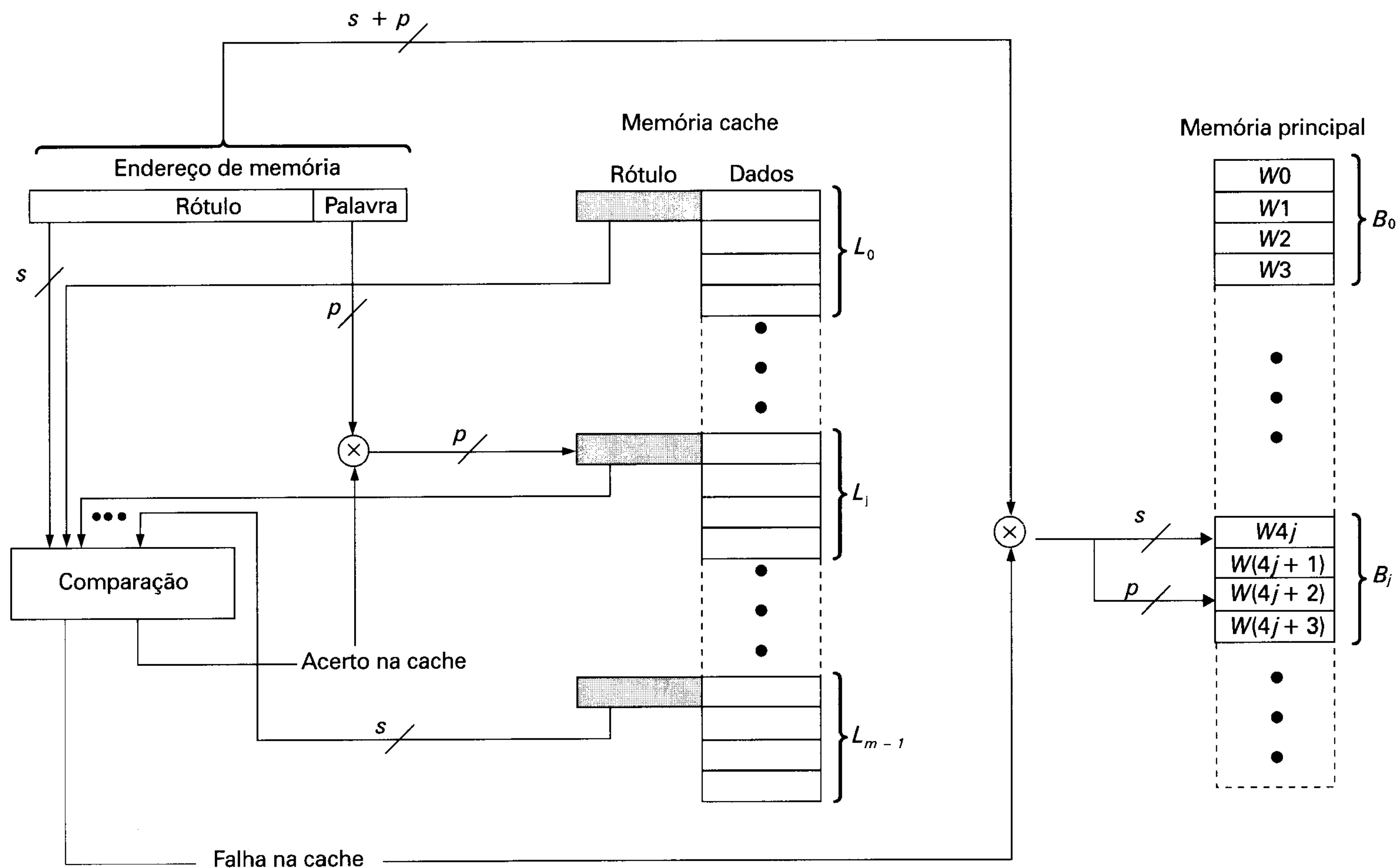
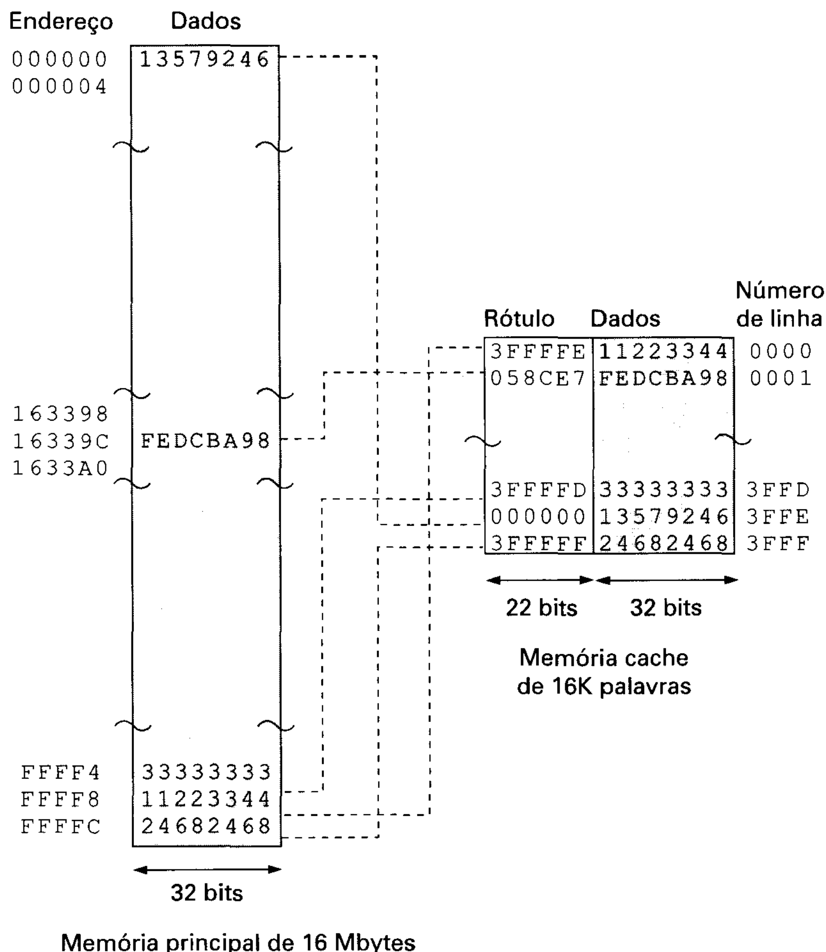


Figura 4.19 Organização de memória cache totalmente associativa (Hwang, 1993).



	Rótulo	Palavra
Endereço de memória principal =	22	2

Figura 4.20 Exemplo de mapeamento associativo.

O mapeamento associativo oferece maior flexibilidade para a escolha do bloco a ser substituído quando um novo bloco é trazido para a memória cache. Algoritmos de substituição, discutidos no decorrer desta seção, são usados para maximizar a taxa de acertos na cache. A principal desvantagem do mapeamento associativo é a complexidade do conjunto de circuitos necessários para a comparação dos rótulos de todas as linhas da memória cache em paralelo.

O **mapeamento associativo por conjuntos** combina as vantagens do mapeamento direto e do mapeamento associativo e diminui suas desvantagens. Nesse mapeamento, a memória cache é dividida em v conjuntos, cada qual com k linhas. As relações são as seguintes:

$$m = v \times k$$
$$i = j \text{ módulo } v$$

onde:

i = número do conjunto na memória cache
j = número do bloco da memória principal
m = número de linhas da memória cache

Um mapeamento desse tipo é denominado mapeamento associativo por conjuntos de k linhas. Um bloco B^j pode ser mapeado em qualquer das linhas do conjunto i. A lógica de controle da memória cache interpreta um endereço de memória como constituído de três campos: rótulo, conjunto e palavra. Os d bits do campo de conjunto especificam um dos $v = 2^d$ conjuntos. Os s bits que compõem os campos de rótulo e de conjunto especificam um dos 2^s blocos da memória principal. A Figura 4.21 mostra a lógica de controle da memória cache. Em um mapeamento totalmente associativo, o rótulo de um endereço de memória é muito grande e é comparado com o rótulo de cada linha de memória cache. Em um mapeamento associativo por conjuntos de k linhas, o rótulo de um endereço de memória é muito menor e é comparado apenas com k rótulos de um mesmo conjunto.

A Figura 4.22 mostra nosso sistema utilizando um mapeamento associativo por conjuntos com duas linhas em cada conjunto, denominado mapeamento associativo por conjuntos de duas linhas. Um número de conjunto de 13 bits identifica um único conjunto (de duas linhas) da memória cache. Ele fornece também um número de bloco da memória principal, módulo 2^{13}. Isso determina o mapeamento de blocos sobre as linhas da memória cache. Assim, os blocos 000000, 008000, ..., FF8000 são mapeados no conjunto 0 da memória cache. Qualquer um desses blocos pode ser carregado em qualquer uma das duas linhas do conjunto. Note que dois blocos mapeados em um mesmo conjunto da memória cache devem possuir rótulos distintos. Em uma operação de leitura, os 13 bits do número de conjunto são utilizados para determinar o conjunto da memória cache a ser examinado. As duas linhas desse conjunto são examinadas pelo campo de rótulo do endereço de memória acessado.

No caso extremo, ou seja, se $v = m$ e $k = 1$, a técnica de mapeamento associativo por conjuntos se reduz ao mapeamento direto. Para $v = 1$ e $k = m$, ela se reduz ao mapeamento associativo. A organização de mapeamento associativo por conjuntos mais comum é a que usa duas linhas por conjunto ($v = m/2$, $k = 2$). Sua taxa de acertos é significativamente maior do que no caso do mapeamento direto. Um mapeamento associativo por conjuntos de quatro linhas ($v = m/4$, $k = 4$) apresenta uma pequena melhoria a um custo adicional relativamente pequeno (Mayberry, 1984; Hill, 1989). Um maior número de linhas por conjunto não apresenta melhoras significativas de desempenho.

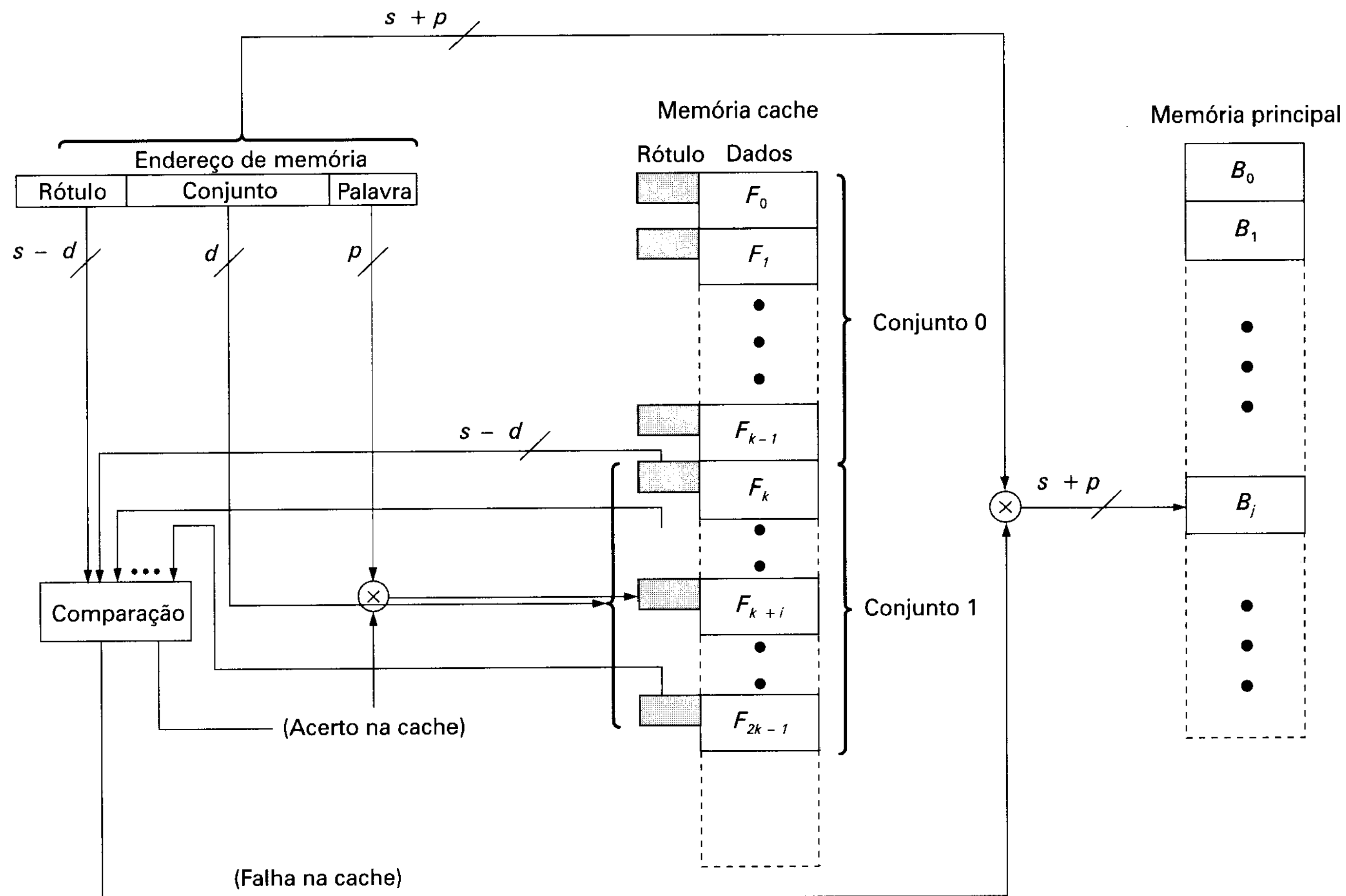


Figura 4.21 Organização de memória cache com mapeamento associativo por conjuntos de k linhas (Hwang, 1993).

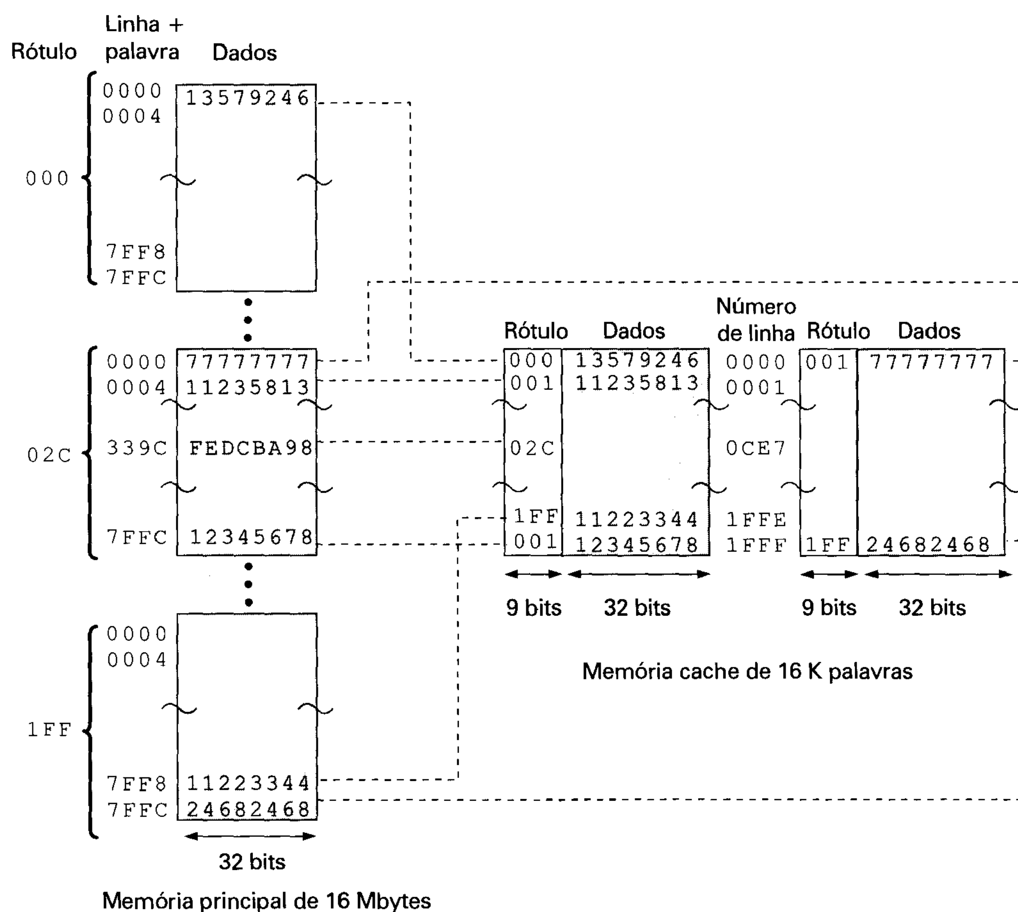


Figura 4.22 Exemplo de mapeamento associativo por conjuntos de duas linhas.

Algoritmos de substituição

Quando um novo bloco é trazido para a memória cache, um dos blocos existentes deve ser substituído. No mapeamento direto, cada bloco é mapeado em uma única linha, o que determina o bloco a ser substituído, não havendo alternativa possível. O mapeamento associativo e o mapeamento associativo por conjuntos requerem o uso de um algoritmo de substituição. Para que a velocidade de acesso à memória seja alta, esse algoritmo deve ser implementado em hardware. Os quatro algoritmos de substituição mais comuns são descritos a seguir. O algoritmo mais eficaz é, provavelmente, o baseado na política de substituir o bloco menos recentemente usado (*least recently used* — LRU): o bloco a ser substituído é o que está no conjunto que não é usado há mais tempo. Esse algoritmo pode ser facilmente implementado no caso de um mapeamento associativo por conjuntos de duas linhas. Cada linha inclui um bit adicional, chamado de bit de USO. Quando uma linha é referenciada, é atribuído valor

1 ao seu bit de USO e o bit de USO da outra linha do conjunto recebe valor 0. Quando um novo bloco deve ser armazenado no conjunto, ele ocupa a linha cujo bit de USO tem valor 0. Supondo que as posições de memória mais recentemente usadas tenham maior probabilidade de ser referenciadas em um futuro próximo, o algoritmo LRU deve fornecer a melhor taxa de acerto. Outra possibilidade é substituir o bloco do conjunto que foi armazenado primeiro na memória cache (*First-in-first-out* — FIFO, ou seja, o primeiro a chegar é o primeiro a sair): o bloco a ser substituído é o que está no conjunto há mais tempo. O algoritmo FIFO pode ser facilmente implementado empregando-se a técnica de área de armazenamento circular. Uma terceira possibilidade é substituir o bloco do conjunto menos freqüentemente utilizado (*least frequently used* — LFU): o bloco a ser substituído é o que foi utilizado menos vezes. O algoritmo LFU pode ser implementado associando-se um contador a cada linha da memória cache. Outra técnica não é baseada no histórico de uso das linhas da memória cache e substitui aleatoriamente uma das linhas candidatas. Estudos baseados em simulação mostram que a substituição aleatória apresenta um desempenho apenas levemente inferior ao de um algoritmo baseado no histórico de uso das linhas (Smith, 1982).

Políticas de atualização

Antes que um bloco residente na memória cache possa ser substituído, é necessário verificar se ele foi alterado na memória cache, mas não na memória principal. Se isso não ocorreu, então o novo bloco pode ser escrito sobre o bloco antigo. Caso contrário, se pelo menos uma operação de escrita foi feita sobre uma palavra dessa linha da memória cache, então a memória principal deve ser atualizada. Podem ser adotadas diferentes políticas de atualização, com diferentes relações de custo e desempenho. Dois problemas devem ser considerados. Primeiramente, a memória principal pode ser utilizada tanto pelo processador quanto pelos dispositivos de E/S (um dispositivo de E/S pode ser capaz de ler/escrever diretamente na memória). Se uma palavra for alterada apenas na memória cache, a palavra de memória correspondente deve ser invalidada. Além disso, se um dispositivo de E/S alterar a memória principal, a palavra correspondente na memória cache deve ser invalidada. O problema é ainda mais complexo se houver múltiplos processadores conectados ao mesmo barramento e cada processador tiver sua própria memória cache local. Nesse caso, se uma palavra for alterada em uma memória cache, isso deve fazer com que ocorra a invalidação de uma palavra em outras memórias cache.

A técnica de atualização mais simples é denominada *escrita direta* (*write-through*). Nessa técnica, todas as operações de escrita são feitas tanto na memória cache quanto na memória principal, assegurando assim que a memória principal esteja sempre com os dados válidos. Qualquer outro módulo composto de um processador e uma memória cache pode monitorar o tráfego para a memória principal, de modo que mantenha a coerência de sua própria memória cache. A principal desvantagem dessa técnica é que ela gera um tráfego de memória considerável, podendo criar um gargalo no sistema. Uma técnica alternativa, conhecida como *escrita de volta* (*write-back*), visa minimizar o número de operações de escrita na memória. Nessa técnica, as escritas são feitas apenas na memória cache. Quando é feita uma atualização, é atribuído o valor 1 a um bit de ATUALIZAÇÃO, associado à linha atualizada na memória cache. Quando um bloco vai ser substituído, ele apenas é escrito de volta na memória principal se seu bit de ATUALIZAÇÃO tiver valor 1. O problema dessa técnica é que partes da memória principal podem ficar inválidas e, portanto, o acesso à memória pelos módulos de

E/S só pode ser efetuado por meio da memória cache. Isso requer um conjunto de circuitos mais complexos e pode também criar um gargalo no sistema. A experiência mostra que as operações de escrita constituem aproximadamente 15% das referências à memória (Smith, 1982).

Em uma organização de sistema com barramento na qual a memória principal é compartilhada e mais de um dispositivo (tipicamente um processador) possui sua própria memória cache, um novo problema é introduzido. Se os dados em uma memória cache forem alterados, não apenas a palavra correspondente na memória principal ficará inválida, mas também as palavras correspondentes nas demais memórias cache (caso alguma outra memória cache contenha a mesma palavra). Mesmo com uma política de escrita direta, as demais memórias cache podem ficar com dados inválidos. Um sistema que evita esse problema é dito um sistema que mantém a coerência de memórias cache. Algumas das abordagens possíveis para manter a coerência de memórias cache são:

- **Monitoramento do barramento com escrita direta:** cada controlador de memória cache monitora as linhas de endereço para detectar operações de escrita na memória feitas por outros mestres do barramento. Caso outro mestre escreva em uma posição da memória compartilhada que também esteja armazenada na sua memória cache, o controlador invalida a entrada correspondente na sua memória cache. Essa estratégia requer o uso de escrita direta por todos os controladores de memórias cache.
- **Transparência em hardware:** um hardware adicional é utilizado para assegurar que todas as atualizações feitas na memória principal por meio de uma memória cache sejam refletidas em todas as demais memórias cache. Dessa maneira, se um processador modifica uma palavra em sua memória cache, além dessa atualização ser feita na memória principal, as palavras correspondentes nas demais memórias cache são também atualizadas.
- **Memória não-cacheável:** apenas uma parte da memória principal é compartilhada por mais de um processador e não pode ser associada à memória cache (não-cacheável). Nesse sistema, todos os acessos à memória compartilhada ocasionam uma falha na memória cache, uma vez que nenhuma posição da memória compartilhada é copiada na memória cache. A porção de memória não-cacheável pode ser identificada utilizando uma lógica de seleção de pastilha ou os bits mais significativos do endereço.

O problema de coerência de memórias cache ainda constitui um tópico de pesquisa, merecendo uma abordagem mais detalhada no Capítulo 16.

Tamanho da linha

Outro elemento de projeto de memórias cache é o tamanho da linha. Quando um bloco de dados é trazido da memória principal para a memória cache, não apenas a palavra requerida é armazenada na memória cache, como também algumas palavras adjacentes. À medida que se amplia o tamanho do bloco, a taxa de acerto na memória cache inicialmente aumenta, em razão do princípio de localidade de referências, segundo o qual a vizinhança de uma palavra que acabou de ser usada tem grande probabilidade de ser utilizada em um futuro próximo. Com um tamanho de bloco maior, mais dados úteis são trazidos para a memória cache.

Entretanto, a taxa de acerto tende a diminuir se o tamanho do bloco se torna tão grande que a probabilidade de utilizar os dados buscados recentemente se torna menor do que a probabilidade de reutilizar os dados que foram substituídos. Dois efeitos específicos devem ser considerados:

- Para uma dada capacidade, o uso de blocos maiores reduz o número de blocos contidos na memória cache. Como cada novo bloco trazido para a memória cache é escrito sobre um bloco anteriormente armazenado, um pequeno número de blocos faz com que os dados sejam sobrescritos tão logo sejam buscados.
- Escolhendo-se um bloco de maior tamanho, cada palavra adicional está mais distante da palavra usada e, portanto, tem menor probabilidade de ser utilizada em um futuro próximo.

A relação entre tamanho de bloco e taxa de acerto é bastante complexa, dependendo das características de localidade de referências de cada programa; nenhum valor ótimo definitivo ainda foi determinado. Um tamanho de bloco de duas a oito palavras parece estar razoavelmente próximo do ótimo (Smith, 1987; Przybylski, 1988 e 1990; Handy, 1998).

Número de memórias cache

Quando as memórias cache foram inicialmente introduzidas, um sistema de memória típico possuía uma única memória cache. Mais recentemente, o uso de várias memórias cache tornou-se comum. Dois aspectos importantes do projeto de sistemas com múltiplas memórias cache são o número de níveis e o uso de memórias cache unificadas ou separadas.

Com o aumento da densidade dos circuitos integrados, foi possível incluir a memória cache na mesma pastilha do processador. Em comparação com uma memória cache conectada ao processador por meio de um barramento externo, a memória cache interna à pastilha (*on-chip*) reduz a atividade do processador no barramento externo e, portanto, diminui o tempo de execução e aumenta o desempenho global do sistema. Quando a instrução ou o valor requerido está na memória cache interna à pastilha, o acesso ao barramento é eliminado. Como os caminhos de dados internos ao processador são bem mais curtos que os feitos por meio de barramentos, o acesso a uma memória cache interna à pastilha é razoavelmente mais rápido, mesmo se considerarmos um ciclo de barramento em que o estado de espera tem tempo nulo. Além disso, durante esse período o barramento fica livre para efetuar outras transferências.

Com a possibilidade de inclusão da memória cache na pastilha do processador, seria ainda desejável utilizar uma cache externa à pastilha? A resposta a essa questão é normalmente positiva: os projetos mais modernos incluem tanto uma memória cache externa quanto outra embutida na pastilha. A organização resultante é conhecida como memória cache de dois níveis, sendo a interna chamada de nível 1 (L1) e a externa, de nível 2 (L2). A razão de incluir uma memória cache L2 é a seguinte. Se a cache L2 não existisse, o acesso à memória DRAM ou ROM seria realizado por meio do barramento sempre que o processador fizesse referência a uma posição de memória que não está na memória cache L1. Por um lado, a velocidade relativamente baixa do barramento e o alto tempo de acesso à memória principal resultam em baixo desempenho; por outro lado, se for usada uma cache L2 com SRAM (RAM estática), os dados que ocasionarem falhas na memória cache L1 freqüentemente poderão ser obtidos ra-

pidamente. Se a memória SRAM for suficientemente rápida em relação à velocidade do barramento, os dados poderão ser obtidos por intermédio de uma transação com tempo de espera nulo, que constitui o tipo mais rápido de transferência de dados por meio do barramento.

A economia de tempo de acesso com o uso de uma memória cache L2 depende das taxas de acerto nas caches L1 e L2. Diversos estudos mostram que, de modo geral, o uso de uma cache de nível 2 de fato melhora o desempenho (veja, por exemplo, Azimi, 1992; Novitsky, 1993; e Handy, 1998).

Quando a memória cache interna à pastilha do processador foi lançada, a maioria dos projetos de memória incluía uma única memória cache para armazenar as referências a dados e instruções. Mais recentemente, tornou-se comum o uso de duas memórias cache: uma dedicada para dados e outra para instruções.

As vantagens potenciais do uso de uma memória cache unificada são as seguintes:

- Para um dado tamanho da memória cache, a taxa de acerto da memória cache unificada é em geral maior do que a de memórias cache separadas, pois ela fornece um balanceamento automático entre as buscas de dados e de instruções. Em outras palavras, se um padrão de execução envolve um número muito maior de buscas de instruções do que de buscas de dados, a memória cache tende a ser preenchida com instruções; se o padrão envolve maior número de busca de dados, ocorre o contrário.
- Apenas uma memória cache precisa ser projetada e implementada.

Apesar dessas vantagens, a tendência atual é utilizar memórias cache separadas para instruções e dados, particularmente em máquinas superescalares, como o Pentium II e o PowerPC. Esses processadores se caracterizam pela execução de instruções em paralelo e pela busca antecipada de futuras instruções. A principal vantagem de um projeto com memórias cache separadas é eliminar a disputa por acesso à memória cache entre o processador de instruções e a unidade de execução. Isso é extremamente importante em projetos que utilizam uma *pipeline* de instruções. Nesse mecanismo, o processador busca instruções a serem executadas antecipadamente, armazenando-as em uma área de armazenamento temporário ou *pipeline*. Suponha que exista uma memória cache unificada para instruções e dados. Quando a unidade de execução de instruções efetua uma operação de leitura ou de escrita de dados, a operação é realizada sobre essa única memória cache. Se, ao mesmo tempo, o mecanismo de busca antecipada de instruções solicitar a busca de uma nova instrução, essa requisição permanece bloqueada até que a memória cache atenda à unidade de execução, possibilitando que a execução da instrução corrente seja completada. Essa disputa por acesso à memória cache interfere bastante na utilização eficiente da *pipeline* de instruções, podendo degradar o desempenho do sistema. Essa dificuldade é superada com uma estrutura de caches separadas.

4.4 ORGANIZAÇÕES DAS MEMÓRIAS CACHE DO PENTIUM II E DO PowerPC

Organização das memórias cache do Pentium II

A evolução da organização de memórias cache pode ser claramente observada por meio da evolução dos microprocessadores da Intel. A pastilha do 80386 não inclui memória cache. A pastilha do 80486 inclui uma memória cache unificada de 8 Kbytes, com uma linha de 16 bytes e mapeamento associativo por conjuntos de quatro linhas. A pastilha do Pentium inclui duas memórias cache, uma para dados e outra para instruções. Cada memória cache tem 8 Kbytes, uma linha de 32 bytes com tamanho e organização associativa por conjuntos de duas linhas. As pastilhas do Pentium Pro e do Pentium II também incluem duas memórias cache L1. As primeiras versões do processador incluem uma memória cache de instruções de 8 Kbytes, com mapeamento associativo por conjuntos de quatro linhas, e outra de dados de 8 Kbytes, com mapeamento associativo por conjuntos de duas linhas. O Pentium Pro e o Pentium II incluem também uma memória cache L2 que alimenta as memórias cache L1. A memória cache L2 é uma memória cache com mapeamento associativo por conjuntos de quatro linhas, com capacidade variando de 256 Kbytes a 1 Mbyte*.

A Figura 4.23 apresenta uma visão simplificada da organização do Pentium II, que mostra o uso de três memórias cache. O núcleo do processador é composto de quatro componentes principais:

- **Unidade de busca/decodificação:** busca as instruções do programa para a memória cache de instruções L1 e decodifica as instruções como uma série de microoperações, que são armazenadas na área de armazenamento temporário de instruções (*instruction pool*).
- **Área de armazenamento temporário de instruções:** armazena um conjunto de instruções disponíveis para execução.
- **Unidade de despacho/execução:** ordena as microoperações para execução, levando em conta as dependências de dados e a disponibilidade de recursos. As microoperações podem ser executadas em uma ordem diferente daquela em que foram buscadas na memória. Quando há tempo disponível, essa unidade executa antecipadamente microoperações que podem ser requeridas no futuro. A execução das microoperações é realizada por essa unidade, buscando os dados requeridos na memória cache de dados L1 e armazenando os resultados em registradores temporários.
- **Unidade de confirmação:** determina quando os resultados armazenados em registradores temporários devem ser confirmados como resultados permanentes e armazenados nos registradores ou na memória cache de dados L1. Depois de confirmar esses resultados, a unidade remove as instruções correspondentes da área de armazenamento temporário de instruções.

---

* N.R.T.: O Intel Pentium 4 inclui uma memória cache L1 de dados de 8 Kbytes e uma memória cache L2 de 256 Kbytes, com mapeamento associativo por conjuntos de oito linhas. Além disso, o Pentium 4 introduz uma memória cache L1 de 12 Kbytes para armazenar as microoperações resultantes da decodificação de instruções recentes, otimizando a execução das instruções.

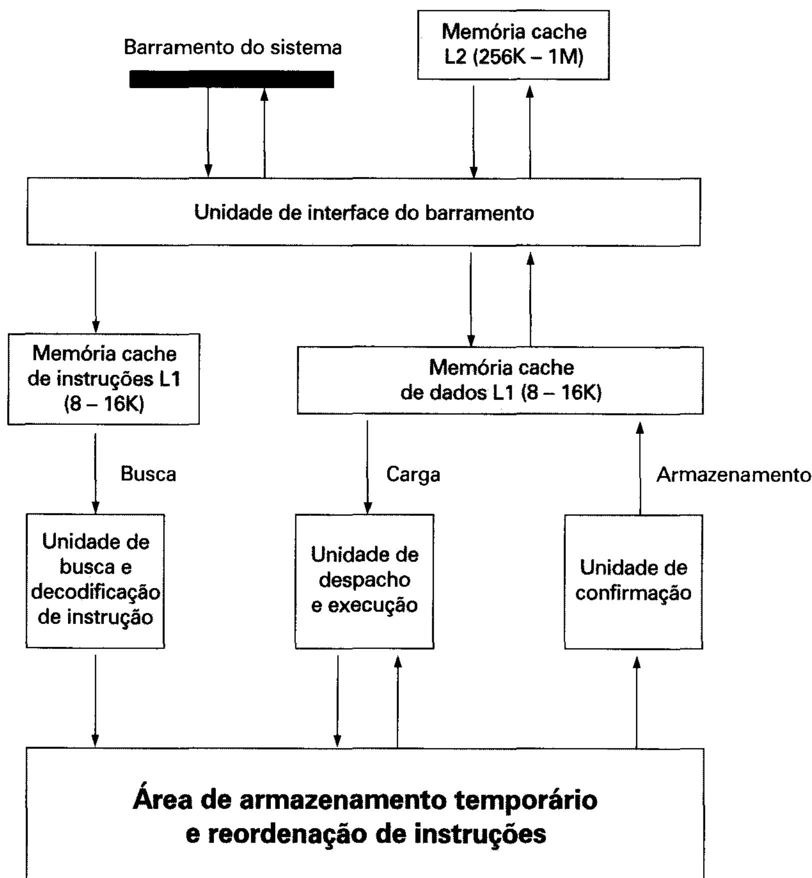
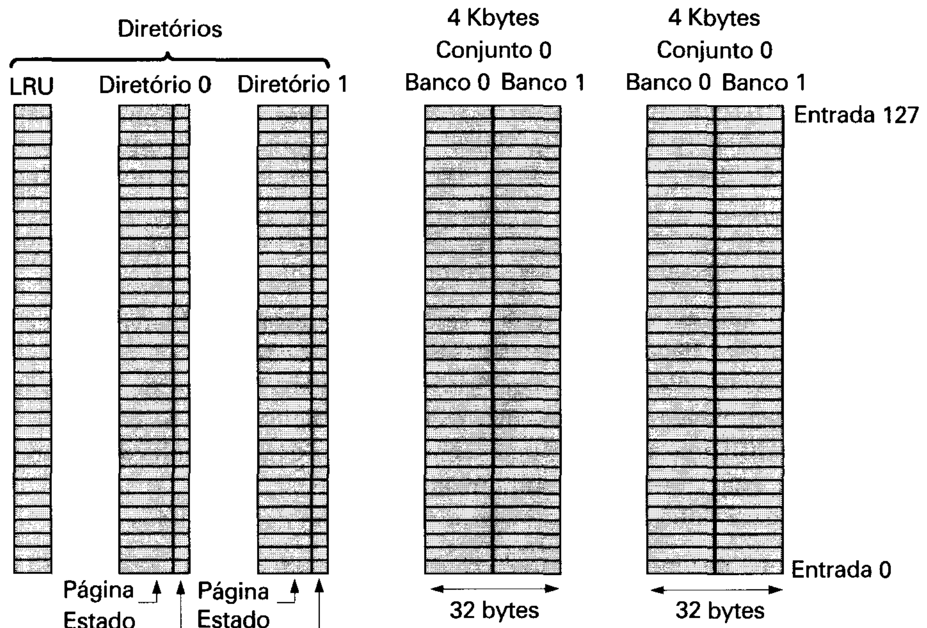


Figura 4.23 Diagrama de blocos do Pentium II.

A Figura 4.24 mostra os principais elementos da memória cache de dados L1. Os dados da memória cache são constituídos de 128 conjuntos, cada qual com duas linhas, logicamente organizados como dois "conjuntos" de 4 Kbytes. A cada linha é associado um rótulo e dois bits de estado; as linhas são logicamente organizadas em dois diretórios, de modo que existe uma entrada de diretório para cada linha da memória cache. O rótulo compõe-se dos 24 bits mais significativos do endereço de memória dos dados armazenados na linha correspondente. O controlador da memória cache utiliza um algoritmo de substituição LRU e, portanto, um único bit de uso (bit LRU) é associado a cada conjunto de duas linhas.

A memória cache de dados utiliza a política de escrita de volta (*write-back*): os dados são gravados na memória principal apenas quando removidos da memória cache e modificados. O processador Pentium II pode ser dinamicamente configurado para usar escrita direta (*write-through*).



	35 ... 11	10 ... 5	4 ... 0
Endereço físico da memória principal =	página (rótulo), 25 bits, 1 de 32M	Linha (1 de 64)	Byte inicial

Figura 4.24 Estrutura da memória cache de dados do Pentium II (Anderson, 1998a).

Tabela 4.5 Estados de linhas da memória cache para o protocolo MESI

	M Modificada	**E Exclusiva**	**C* Compartilhada**	**I Inválida**
A linha da memória cache é válida?	Sim	Sim	Sim	Não
A cópia na memória está...	Desatualizada	Válida	Válida	—
Existem cópias em outras memórias cache?	Não	Não	Talvez	Talvez
Uma operação de escrita nessa linha...	Não vai para o barramento	Não vai para o barramento	Vai para o barramento e atualiza a memória cache	Vai diretamente para o barramento

* N.P.: Do original: *S — Shared*.

Coerência de memórias cache de dados

O protocolo MESI (*modified/exclusive/shared/invalid* — modificado / exclusivo / compartilhado / inválido) é utilizado em memórias cache de dados para assegurar a coerência dos dados. Ele é projetado para atender aos requisitos de coerência de dados em sistemas com vários processadores, além de ser útil em uma organização com um único processador como a do Pentium II.

A memória cache de dados inclui dois bits de estado por rótulo, de modo que cada linha pode estar em um dos quatro estados seguintes:

- **Modificada:** a linha na memória cache foi modificada (é diferente da memória principal) e está disponível apenas nessa memória cache.
- **Exclusiva:** a linha de uma memória cache é igual à linha da memória principal e não está presente em nenhuma outra memória cache.
- **Compartilhada:** a linha de uma memória cache é igual à linha da memória principal e pode estar presente em outra memória cache.
- **Inválida:** a linha da memória cache não contém dados válidos.

A Tabela 4.5 resume o significado dos quatro estados. O protocolo MESI é discutido mais detalhadamente no Capítulo 16.

Controle de memórias cache

Uma memória cache interna é controlada por dois bits em um dos registradores de controle, denominados bits CD (cache desabilitada) e NW (não escrita direta) descritos na Tabela 4.6. Duas instruções do Pentium II são utilizadas para controlar a memória cache: a instrução INVD invalida os dados da memória cache interna e envia um sinal para a memória cache externa (se existir) indicando que seus dados devem ser invalidados. A instrução WBINVD executa a operação de *write-back* da memória cache interna e invalida seus dados e então executa a operação de *write-back* para a memória cache externa e invalida seus dados.

Tabela 4.6 Modos de operação da memória cache do Pentium II

Bits de controle		Modo de operação		
CD	NW	Entrada na memória cache	Escrita direta	Invalidação
0	0	Habilitada	Habilitada	Habilitada
1	0	Desabilitada	Habilitada	Habilitada
1	1	Desabilitada	Desabilitada	Desabilitada

Nota: CE = 0; NW = 1 é uma combinação inválida.

Organização das memórias cache do PowerPC

A organização do sistema de memórias cache do PowerPC evoluiu juntamente com toda a arquitetura da família de processadores PowerPC, refletindo a incansável busca por melhor desempenho que é o objetivo de todo projetista de microprocessadores.

A Tabela 4.7 mostra essa evolução. O modelo original, o processador 601, inclui uma única memória cache de dados e instruções, de 32 Kbytes, com mapeamento associativo por conjuntos de oito linhas. O processador 603 tem um projeto RISC mais sofisticado, mas inclui

uma memória cache menor: 16 Kbytes divididos em memórias cache separadas para instruções e dados, ambas organizadas com mapeamento associativo por conjuntos de duas linhas. O resultado é que o 603 tem aproximadamente o mesmo desempenho do 601, a um custo mais baixo. Tanto o 604 quanto o 620 possuem memórias cache com o dobro do tamanho do modelo anterior. O modelo atual, o G3, possui memórias cache L1 de tamanho igual aos do 620*.

A Figura 4.25 apresenta uma visão simplificada da organização do PowerPC G3, com enfoque nas duas memórias cache; a organização dos demais membros da família é similar. As unidades de execução centrais são as duas unidades lógica e aritmética de inteiros, que podem operar paralelamente, e uma unidade de ponto flutuante, que possui seus próprios registradores e circuitos de multiplicação, adição e divisão. A memória cache de dados alimenta as operações sobre números inteiros e de ponto flutuante, por meio de uma unidade de carga e armazenamento. A memória cache de instruções, apenas de leitura, alimenta uma unidade de instrução cuja operação é discutida no Capítulo 13.

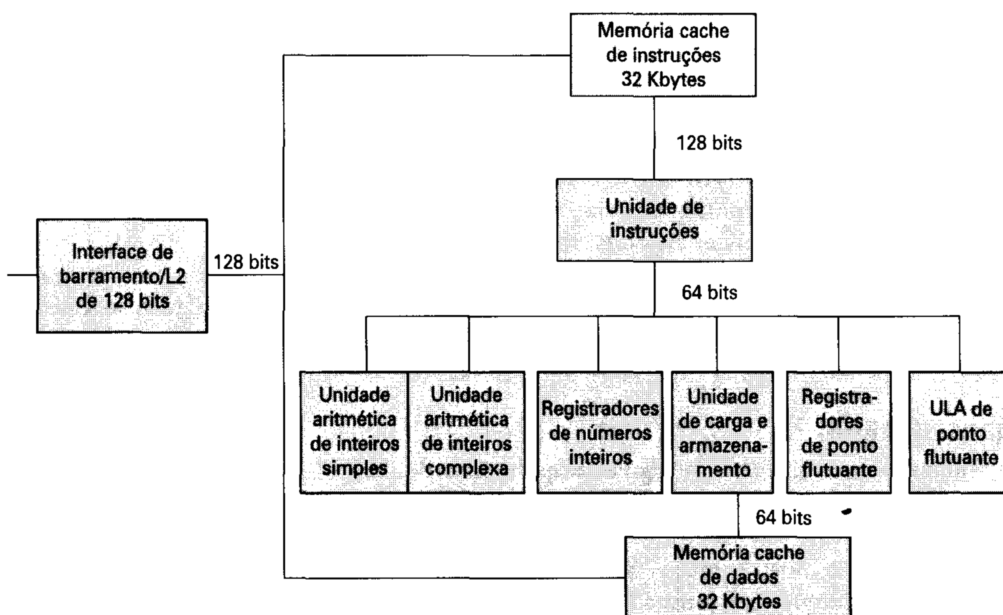


Figura 4.25 Diagrama de blocos do PowerPC G3.

As memórias cache L1 são caches com mapeamento associativo por conjuntos de oito linhas e utilizam uma versão do protocolo de coerência MESI. A memória cache L2 é uma cache com mapeamento associativo por conjuntos de duas linhas, com 256K, 512K ou 1 Mbyte de memória.

* N.R.T.: O mais recente modelo da família PowerPC, o G4, possui memória cache L1 de dados e de instruções de 32 Kbytes cada um, com linhas de 32 bytes e mapeamento associativo por conjuntos de oito linhas. O G4 inclui também uma memória cache L2 de 256 Kbytes a 2 Mbytes, com linhas de 64 bytes e mapeamento associativo por conjuntos de duas linhas.

Tabela 4.7 Memórias cache internas do PowerPC

Modelo	Tamanho	Bytes por linha	Organização
PowerPC 601	1 32 Kbytes	32	Associativo por conjuntos de 8 linhas
PowerPC 603	2 8 Kbytes	32	Associativo por conjuntos de 2 linhas
PowerPC 604	2 16 Kbytes	32	Associativo por conjuntos de 4 linhas
PowerPC 620	2 32 Kbytes	64	Associativo por conjuntos de 8 linhas

4.5 ORGANIZAÇÕES DE DRAM AVANÇADA

Como discutimos no Capítulo 2, um dos pontos mais críticos de sistemas com processadores de alto desempenho é a interface com a memória principal interna. Essa interface é o caminho mais importante de todo o sistema de computação. A pastilha de DRAM permanece como o bloco básico de construção da memória principal e não houve mudança significativa na arquitetura de memórias DRAM desde o início dos anos 70 até recentemente. A pastilha de DRAM tradicional possui restrições devidas à sua arquitetura interna e à sua interface com o barramento de memória do processador.

Vimos anteriormente que uma abordagem para superar o problema de desempenho da memória principal DRAM é inserir um ou mais níveis de memória cache SRAM de alta velocidade, entre a memória principal DRAM e o processador. No entanto, uma memória SRAM é muito mais cara do que uma memória DRAM e o aumento do tamanho da memória cache além de certo ponto traz benefícios desprezíveis.

Nos últimos anos, foram exploradas diversas técnicas para melhorar a arquitetura básica de memórias DRAM, algumas delas atualmente disponíveis no mercado. Não se sabe ainda qual dessas técnicas permanecerá como um padrão de DRAM ou mesmo quais delas sobreviverão. Esta seção apresenta uma visão geral dessas novas tecnologias de DRAM.

Enhanced DRAM

Talvez a mais simples das novas arquiteturas de DRAM seja a pastilha de enhanced DRAM (EDRAM) desenvolvida pela Ramtron (Bondurant, 1994). A EDRAM integra uma pequena memória cache SRAM em uma pastilha de DRAM genérica.

A Figura 4.26 mostra uma versão de EDRAM de 4 Mbits. A memória cache SRAM armazena todo o conteúdo da última linha lida, que é constituído de 2.048 bits ou 512 porções de 4 bits. Um circuito de comparação armazena o valor de 11 bits do endereço de linha mais recentemente selecionado. Se a próxima referência for para a mesma linha, o acesso será feito apenas à memória cache fast SRAM.

A memória EDRAM inclui várias outras características que melhoram seu desempenho. As operações de regeneração de dados podem ser conduzidas em paralelo com operações de leitura na memória cache, minimizando o tempo em que a pastilha não está disponível em razão da regeneração. Note também que o caminho de leitura da linha da memória cache para a porta de saída é independente do caminho de escrita do módulo de E/S para os amplificadores de estado. Isso permite que um acesso de leitura subseqüente à memória cache possa ser efetuado em paralelo enquanto a operação de escrita é completada.

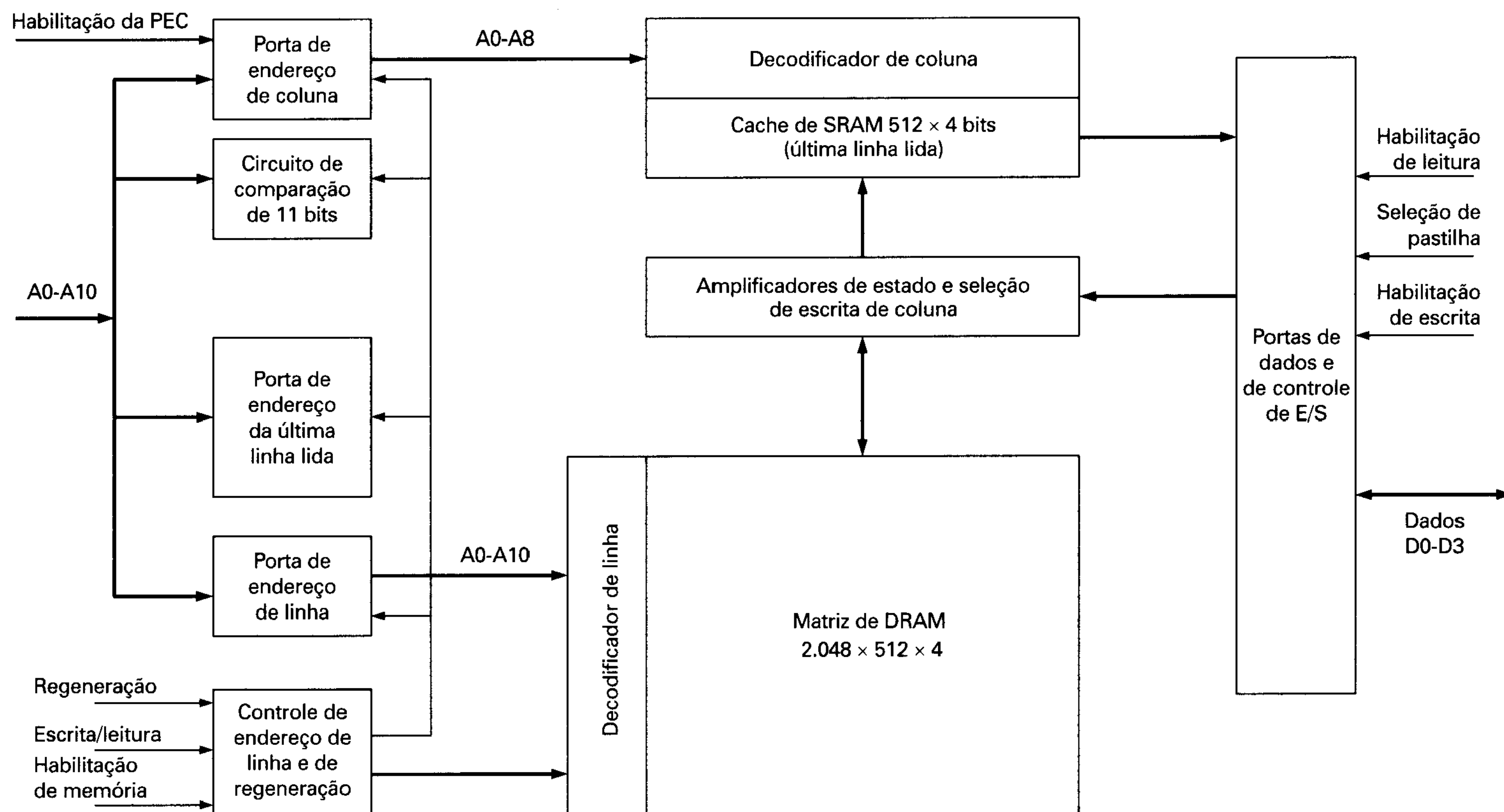


Figura 4.26 Organização da enhanced DRAM (EDRAM).

Estudos feitos pela Ramtron indicam que uma pastilha de memória EDRAM tem desempenho igual ou melhor que o de uma memória DRAM comum combinada com uma memória cache SRAM externa (à DRAM), porém de tamanho maior.

Cache DRAM

A pastilha de memória cache DRAM (CDRAM), desenvolvida pela Mitsubishi (Hidaka, 1990), é similar à EDRAM. Ela inclui uma cache SRAM maior que a da EDRAM (16 *versus* 2 Kbytes).

Na CDRAM, a memória SRAM pode ser usada de dois modos. O primeiro, como uma memória cache propriamente dita, consiste em certo número de linhas de 64 bits. Isso difere da EDRAM, no qual a memória cache SRAM contém apenas um bloco, ou seja, a linha utilizada mais recentemente. O modo de memória cache da CDRAM é eficiente em acessos aleatórios à memória.

A SRAM da CDRAM pode também ser utilizada como área de armazenamento temporário para dar suporte ao acesso seqüencial a um bloco de dados. Por exemplo, na execução de uma operação de restauração de uma imagem no vídeo a partir do seu mapa de bits, a CDRAM pode buscar os dados antecipadamente da área de DRAM para a área de SRAM. Acessos subseqüentes à pastilha resultam em acessos apenas à área de SRAM.

DRAM síncrona

Uma abordagem bastante diferente para obter um melhor desempenho para a DRAM é o uso da DRAM síncrona (SDRAM), que vem sendo desenvolvida em conjunto por diversas companhias (Vogley, 1994).

Diferentemente da memória DRAM típica, que é assíncrona, a troca de dados entre a SDRAM e o processador é sincronizada por um sinal de relógio externo e executada na velocidade do barramento do processador e da memória, sem estados de espera.

Em uma pastilha de DRAM típica, o processador fornece à memória um endereço e sinais de controle, indicando que um conjunto de dados em uma determinada posição de memória deve ser lido ou escrito na DRAM. Após certo atraso, correspondente ao tempo de acesso, a DRAM efetua a leitura ou a escrita dos dados. Durante esse tempo de acesso, a pastilha de DRAM executa várias funções internas, como ativar uma capacitância alta nas linhas de sinal de linha e de coluna da memória cache, ler os dados e enviá-los por meio da área de armazenamento temporário de saída. O processador simplesmente tem de esperar por esse atraso, diminuindo assim o desempenho do sistema.

Com o acesso síncrono, a pastilha de DRAM transfere os dados sob o controle do relógio do sistema. O processador ou qualquer outro mestre do barramento envia uma instrução e os dados de endereçamento para a DRAM, que responde após um determinado número de ciclos de relógio. Enquanto a SDRAM está processando a requisição, o mestre de barramento pode executar outras tarefas.

A Figura 4.27 mostra a lógica interna de uma pastilha de memória SDRAM. A SDRAM utiliza um modo de transferência de dados em seqüência (burst mode), que elimina o tempo requerido para obter o endereço e para carregar os endereços de linha e de coluna da memória cache, nos acessos subseqüentes ao primeiro. Nesse modo, uma série de bits de dados pode ser transferida rapidamente depois de obtido o primeiro bit. Esse modo de transferência de dados é útil quando os bits a serem transferidos estão em seqüência e na mesma linha na matriz do bit inicial.

A memória SDRAM utiliza, além disso, uma arquitetura interna com dois bancos de dados, o que aumenta as oportunidades de paralelismo interno na pastilha.

Figura 4.27 Organização da memória DRAM síncrona (SDRAM) (Przybylski, 1994).

Outra característica fundamental que diferencia a SDRAM de uma DRAM convencional é o registrador de modo e a lógica de controle associada, que fornecem um mecanismo para adequar a SDRAM às necessidades específicas do sistema. O registrador de modo especifica o tamanho da seqüência de dados, ou seja, o número de unidades de dados a serem transferidas sincronamente por meio do barramento. Esse registrador também permite ao programador ajustar o tempo de latência entre o recebimento de um requisito de leitura e o início da transferência de dados.

A memória SDRAM oferece um melhor desempenho para a transferência serial de grandes blocos de dados, tal como em aplicações de processamento de texto e multimídia*.

---

* N.R.T.: Atualmente existe uma evolução da SDRAM denominada DDR-SDRAM (Double Data Rate SDRAM). A DDR-SDRAM permite a transferência de dados nas duas bordas do sinal de relógio do sistema, obtendo o dobro do desempenho alcançado por uma SDRAM.

Rambus DRAM

A pastilha RDRAM, desenvolvida pela Rambus (Garrett, 1994; Crisp, 1997), adota uma abordagem mais revolucionária para o problema da largura de banda de memória. As pastilhas RDRAM são empacotadas verticalmente, com todos os pinos de um mesmo lado. A pastilha troca dados com o processador por meio de 28 fios de menos de 12 cm de comprimento. O barramento pode endereçar até 320 pastilhas RDRAM, transferindo dados a uma taxa de 500 Mbps. Essa taxa é comparável à taxa de cerca de 33 Mbps de uma DRAM assíncrona.

O barramento especial RDRAM utiliza um protocolo assíncrono orientado a blocos para o envio de endereço e informações de controle. Após um tempo de acesso inicial de 480 ns, a taxa de transferência de dados é de 500 Mbps. O que torna essa velocidade possível é o próprio barramento, que define impedâncias, pulsos de relógio e sinais de maneira bastante precisa. Em vez de ser explicitamente controlada pelos sinais RAS, CAS, R/W e CE, utilizados nas DRAMs convencionais, uma RDRAM recebe requisições de acesso à memória por meio do barramento de alta velocidade. Uma requisição contém o endereço desejado, o tipo da operação e o número de bytes a serem transferidos.

RamLink

A mudança mais radical em relação à pastilha de DRAM convencional ocorre na RamLink (Gjessing, 1992), desenvolvida pelo grupo de trabalho SCI (*scalable coherent interface* — interface coerente escalável) do IEEE. A RamLink concentra-se na interface entre o processador e a memória e não na arquitetura interna das pastilhas de DRAM.

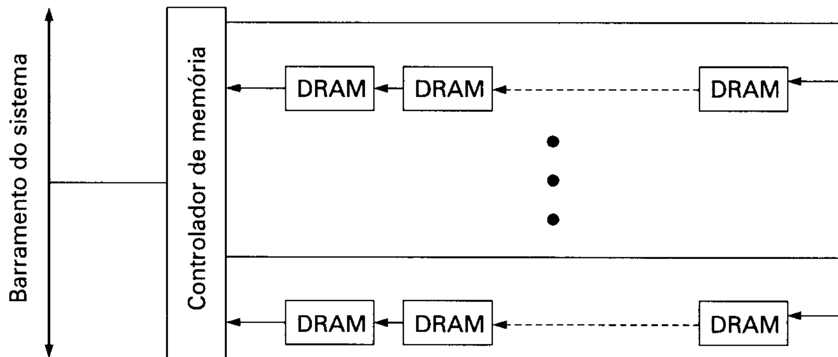
A RamLink consiste em uma interface de memória com conexões ponto a ponto, dispostas em anel (Figura 4.28a). O tráfego no anel é gerenciado por um controlador de memória que envia mensagens às pastilhas de DRAM, as quais atuam como nós na rede em anel. Os dados são transferidos na forma de pacotes (Figura 4.28b).

Os pacotes de requisição iniciam as transações de memória. Eles são enviados pelo controlador e contêm um cabeçalho de comando, um endereço, bits de verificação e, no caso de comandos de escrita, os dados a serem escritos. O cabeçalho de comando é constituído pelo tipo de comando, pelo tamanho dos dados a serem transferidos e pelas informações de controle e contém um tempo de resposta específico ou um tempo máximo permitido para que o escravo envie a resposta. As informações de controle incluem um bit que indica se requisições subseqüentes serão para endereços consecutivos. Até quatro transações podem ser ativadas simultaneamente por dispositivo; portanto, todos os pacotes contêm um identificador de transação (ID) de 2 bits, utilizado para casar os pacotes de requisição e de resposta de maneira não ambígua.

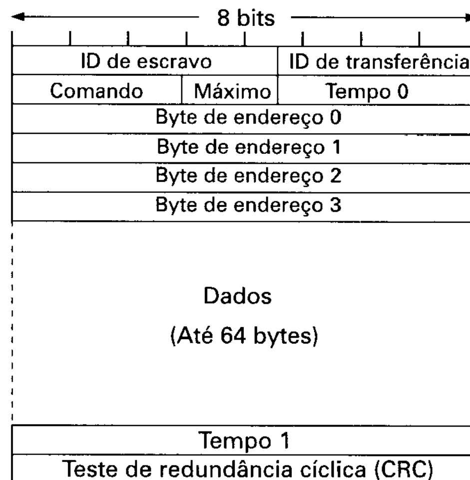
Para que uma operação de leitura seja bem-sucedida, a DRAM escrava envia um pacote de resposta com os dados lidos. Em uma requisição malsucedida, o escravo envia um pacote para nova tentativa, que indica o tempo adicional necessário para completar a transação.

Uma das vantagens da abordagem RamLink é ser uma arquitetura escalável, que possibilita o uso de um número pequeno ou grande de pastilhas de DRAM e impõe a estrutura interna da memória DRAM. O arranjo em anel da RamLink é projetado para coordenar a atividade de várias pastilhas de DRAM e fornecer uma interface eficiente com controlador de memória.

Um melhoramento recente da RamLink que faz uso da tecnologia desenvolvida para a SDRAM é conhecido como SyncLink DRAM ou SLDRAM (Gillingham, 1997).



(a) Arquitetura da RamLink



(b) Formato do pacote

Figura 4.28 RamLink.

4.6 LEITURA E SITES WEB RECOMENDADOS

Uma abordagem bastante abrangente das tecnologias de memórias de semicondutores, incluindo SRAM, DRAM e memórias flash, é apresentada em Prince (1991). Esse mesmo tema é abordado em Sharma (1997), porém com maior ênfase em questões de teste e de confiabilidade. Prince (1996) focaliza principalmente arquiteturas avançadas de DRAM e SRAM.

Uma boa explicação sobre códigos de correção de erros pode ser encontrada em McEliece (1985) e um estudo mais detalhado é apresentado em Adamek (1991) e Blahut (1983). Sharma (1997) apresenta uma boa visão sobre os códigos utilizados em memórias principais mais modernas.

Uma extensiva abordagem sobre projeto de memórias cache é apresentada em Handy (1998). Uma descrição detalhada da organização das memórias cache do Pentium II pode ser

encontrada em Anderson (1998a) e da organização das memórias cache do PowerPC em Motorola, Inc. (1997) e Shanley (1995a). Um artigo clássico que ainda vale a pena ser lido é o de Smith (1982); ele examina os vários elementos de projeto de memórias cache e apresenta os resultados de um extenso conjunto de análises. Um exame detalhado de diversas questões de projeto de memórias cache relacionadas com multiprogramação e multiprocessamento é apresentado em Agarwal (1989). Higbie (1990) fornece um conjunto de fórmulas simples que podem ser utilizadas para estimar o desempenho de memórias cache em função de vários parâmetros.



Site Web recomendado:

- **The RAM guide:** uma boa visão geral sobre a tecnologia de memórias RAM, além de vários endereços úteis.

4.7 EXERCÍCIOS

4.1 Indique as razões pelas quais as memórias RAM têm sido organizadas tradicionalmente apenas com um bit por pastilha, enquanto as memórias ROM têm sido organizadas com vários bits por pastilha.

4.2 Considere uma RAM dinâmica cujos dados devem ser regenerados 64 vezes por milissegundo. Cada operação de regeneração requer 150 ns; um ciclo de memória requer 250 ns. Qual é a porcentagem do tempo total de operação da memória que é consumida na regeneração de dados?

4.3 Projete uma memória de 16 bits, com capacidade total de 8192 bits, usando pastilhas SRAM de tamanho 64×1 bit. Determine a configuração das pastilhas na placa de memória, mostrando todos os sinais de entrada e saída necessários para associar essa memória à parte inferior do espaço de endereçamento. O projeto deve permitir acesso à memória tanto a bytes quanto a palavras de 16 bits.
Fonte: Alexandridis (1993).

4.4 Suponha que uma palavra de dados de 8 bits armazenada na memória tenha conteúdo 11000010. Usando o algoritmo de Hamming, determine os bits de verificação que seriam armazenados na memória com essa palavra de dados. Mostre como você obteve sua resposta.

4.5 Os bits de verificação armazenados com a palavra de 8 bits 00111001 são 0111. Suponha que, quando a palavra é lida da memória, os bits de verificação são calculados como 1101. Qual é a palavra de dados que foi lida da memória?

4.6 Quantos bits de verificação são necessários se o código de correção de erros de Hamming for usado para detectar erro em um único bit de palavras de dados de 1.024 bits?

4.7 Desenvolva um código SEC para uma palavra de dados de 16 bits. Gere o código para a palavra de dados 0101000000111001. Mostre que o código identifica erros corretamente no quarto bit de dados.

4.8 Uma memória cache associativa por conjuntos contém 64 linhas agrupadas em conjuntos de quatro linhas. A memória principal contém 4K blocos de 128 palavras cada um. Mostre o formato dos endereços da memória principal.

4.9 Para os endereços hexadecimais da memória principal 111111, 666666, BBBBBB, mostre as seguintes informações, em formato hexadecimal:

a. Os valores dos campos de rótulo, linha e palavra, para uma memória cache com mapeamento direto, usando o formato da Figura 4.18.

b. Os valores dos campos de rótulo e palavra, para uma memória cache associativa, usando o formato da Figura 4.20.

c. Os valores dos campos de rótulo, conjunto e palavra, para uma memória cache associativa por conjuntos de duas linhas, usando o formato da Figura 4.22.

4.10 Considere um microprocessador de 32 bits, com uma memória cache interna à pastilha de 16 Kbytes, organizada com mapeamento associativo por conjuntos de quatro linhas. Suponha que o tamanho da linha da memória cache seja de quatro palavras de 32 bits. Desenhe um diagrama de blocos dessa memória cache, mostrando sua organização e como os diferentes campos do endereço são usados para determinar um acerto ou falha na memória cache. Onde a palavra de memória de endereço ABCDE8F8 é mapeada na memória cache?
Fonte: Alexandridis (1993).

4.11 Suponha as seguintes especificações para uma memória cache externa: mapeamento associativo por conjuntos de quatro linhas; tamanho de linha igual a duas palavras de 16 bits; capaz de acomodar um total de 4K palavras de 32 bits da memória principal; utilizada com um processador de 16 bits que gera endereços de 24 bits. Projete a estrutura da memória cache com todas as informações pertinentes e mostre como ela interpreta os endereços enviados pelo processador.
Fonte: Alexandridis (1993).

4.12 A pastilha do processador Intel 80486 possui uma memória cache única para dados e instruções. Esse processador tem capacidade de 8 Kbytes e é organizado com mapeamento associativo por conjuntos de quatro linhas e com blocos de quatro palavras de 32 bits. A memória cache é organizada em 128 conjuntos. Existe um único "bit de linha válida" e três bits, B0, B1 e B2 (bits de uso para o algoritmo LRU), por conjunto. No caso de um acesso com falha na cache, o 80486 lê uma linha de 16 bytes da memória principal, em uma única leitura por meio do barramento de memória. Desenhe um diagrama simplificado da memória cache e mostre como os diferentes campos do endereço são interpretados.
Fonte: Alexandridis (1993).

4.13 Considere uma máquina com memória endereçada byte a byte, com tamanho de 2^{16} bytes e tamanho de bloco de 8 bytes. Suponha que seja utilizada uma memória cache com mapeamento direto, composta de 32 linhas.

a. Como o endereço de memória de 16 bits é dividido em rótulo, número de linha e número de byte?

b. Em que linha seriam armazenados os bytes com os seguintes endereços?
0001 0001 0001 1011
1100 0011 0011 0100
1101 0000 0001 1101
1010 1010 1010 1010

c. Suponha que o byte de endereço 0001 1010 0001 1010 esteja armazenado na memória cache. Quais são os endereços dos outros bytes na mesma linha?

d. Qual o total de bytes de memória que podem ser armazenados na memória cache?

e. Por que o rótulo também é armazenado na memória cache?

4.14 O algoritmo de substituição do Intel 486 é conhecido como pseudo LRU. Três bits B0, B1 e B2 são associados a cada um dos 128 conjuntos de quatro linhas (rotuladas L1, L2, L3, L4). O algoritmo de substituição funciona da seguinte maneira: quando uma linha deve ser substituída, a memória cache primeiro determina se o uso mais recente ocorreu em L0 e L1 ou em L2 e L3. Em seguida, a memória cache determina qual dentre o par de blocos foi o bloco menos usado recentemente, selecionando-o para substituição.

a. Especifique como os valores dos bits B0, B1 e B2 são modificados e como eles são utilizados no algoritmo de substituição.

b. Mostre que o algoritmo utilizado no 80486 se aproxima do algoritmo LRU verdadeiro.

c. Mostre que o algoritmo LRU verdadeiro requer 6 bits por conjunto.

4.15 Uma memória cache associativa por conjuntos tem um tamanho de bloco de quatro palavras de 16 bits e um conjunto de duas linhas. A memória cache pode acomodar um total de 4048 palavras. A porção da memória principal que é cache tem dimensão de 64K × 32 bits. Projete a estrutura da memória cache e mostre como os endereços do processador são interpretados.

Fonte: Alexandridis (1993).

4.16 Descreva uma técnica simples para implementar um algoritmo de substituição LRU em uma memória cache associativa por conjuntos de quatro linhas.

4.17 Considere o seguinte código:

```
for (i = 0; i < 20; i++)
     for (j = 0; j < 10; j++)
     a[i] = a[i] * j
```

a. Dê um exemplo de localidade espacial no código.

b. Dê um exemplo de localidade temporal no código.

4.18 Generalize as equações 4.1 e 4.2, do Apêndice 4A, para hierarquias de memória de N níveis.

4.19 Um computador tem uma memória principal com 32K palavras de 16 bits. Tem também uma memória cache de 4K palavras, dividida em conjuntos de quatro linhas com 64 palavras por linha. Suponha que a memória cache esteja inicialmente vazia. O processador busca palavras das posições 0, 1, 2, ..., 4351, nessa ordem. Ele então repete essa seqüência de referências mais nove vezes. A memória cache é dez vezes mais rápida que a memória principal. Estime a melhoria de desempenho obtida com o uso da memória cache. Suponha que seja utilizado o algoritmo LRU para substituição de blocos.

4.20 Considere um sistema de memória com os seguintes parâmetros:

T_c = 100 ns C_c = 0,01 centavo/bit

T_m = 1200 ns C_m = 0,001 centavo/bit

a. Qual é o custo de 1 Mbyte de memória principal?

b. Qual é o custo de 1 Mbyte de memória principal utilizando tecnologia de memória cache?

c. Se o tempo de acesso efetivo é 10% maior que o tempo de acesso à memória cache, qual é a taxa de acerto H?

4.21 Um computador possui uma memória cache, uma memória principal e um disco usado para memória virtual. Se uma palavra referenciada está na memória cache, são necessários 20 ns para obtê-la. Se ela está na memória principal, mas não na memória cache, são necessários 60 ns para carregá-la na memória cache e então a referência é novamente

iniciada. Se ela não está na memória principal, são necessários 12 ms para buscá-la no disco, mais 60 ns para copiá-la na memória cache e então a referência é novamente iniciada. A taxa de acesso com acerto na memória cache é de 0,9 e a na memória principal é de 0,6. Qual é o tempo médio em nanossegundos necessário para acessar uma palavra referenciada nesse sistema?

APÊNDICE 4A CARACTERÍSTICAS DE DESEMPENHO DE MEMÓRIAS DE DOIS NÍVEIS

Neste capítulo, fazemos referências a memórias cache que atuam como área de armazenamento temporário entre a memória principal e o processador, criando uma memória interna de dois níveis. Essa arquitetura de dois níveis apresenta um desempenho melhor do que uma memória de um único nível, explorando a propriedade conhecida como localidade de referências, que é examinada neste apêndice.

O mecanismo de utilização de uma memória cache para armazenar parte do conteúdo da memória principal é parte da arquitetura do computador, implementada em hardware e normalmente é invisível ao sistema operacional. Existem dois outros exemplos do uso de memória de dois níveis, que também exploram a localidade de referências e que são implementados (pelo menos parcialmente) pelo sistema operacional: a memória virtual e a memória cache de disco (Tabela 4.8). A memória virtual é abordada no Capítulo 7; a memória cache de disco está além do escopo deste livro, sendo abordada em Stallings (1998). Neste apêndice, examinamos algumas características de desempenho de memórias de dois níveis que são comuns a essas três abordagens.

Localidade

A base para o melhor desempenho de uma memória de dois níveis é um princípio conhecido como *localidade de referências* (Denning, 1968). Esse princípio estabelece que as referências à memória tendem a se agrupar. Ao longo de um grande período de tempo, os agrupamentos em uso mudam, mas durante períodos curtos de tempo o processador trabalha com agrupamentos fixos de referências à memória.

Tabela 4.8 Características de memórias de dois níveis

	Cache da memória principal	Memória virtual (paginação)	Cache de disco
Taxa de tempos de acesso típicos	5/1	1000/1	1000/1
Sistema de gerenciamento de memória	Implementado por hardware especial	Combinação de hardware e software de sistema	Software de sistema
Tamanho de bloco típico	4 a 128 bytes	64 a 4096 bytes	64 a 4096 bytes
Acesso do processador ao segundo nível	Acesso direto	Acesso indireto	Acesso indireto

Intuitivamente, o princípio de localidade faz sentido. Considere a seguinte linha de raciocínio:

1. Exceto no caso de instruções de desvio e chamadas de sub-rotinas, que constituem uma pequena fração das instruções de um programa, a execução de um programa é seqüencial. Assim, na maioria dos casos, a próxima instrução a ser buscada segue a seqüência das últimas instruções buscadas.
2. É raro existir uma longa seqüência ininterrupta de chamadas de procedimento, seguida pela seqüência correspondente de instruções de retorno. Ao contrário, um programa geralmente permanece confinado a uma janela bastante estreita de chamadas de procedimentos. Portanto, em curtos períodos de tempo as referências às instruções tendem a ficar restritas a poucos procedimentos.
3. A maioria das construções iterativas é composta por um número relativamente pequeno de instruções, que são repetidas várias vezes. Portanto, durante uma iteração, a computação fica confinada a pequenas porções contíguas do programa.
4. Em muitos programas, grande parte da computação envolve o processamento de estruturas de dados, tais como matrizes ou seqüências de registros. Em muitos casos, sucessivas referências a essas estruturas de dados são feitas a itens de dados próximos uns dos outros.

Essa linha de raciocínio foi confirmada por diversos estudos. Em relação ao item 1, vários estudos analisaram o comportamento de programas em linguagem de alto nível. A Tabela 4.9 apresenta os principais resultados de medidas de ocorrência de vários tipos de comandos ao longo da execução de programas, obtidos pelos estudos a seguir. Os primeiros estudos de comportamento de programas em linguagem de alto nível, feitos por Knuth (1971), examinavam uma coleção de programas escritos em FORTRAN propostos como exercícios para os estudantes. Tanenbaum (1978) publicou dados referentes à execução de mais de 300 procedimentos usados em sistemas operacionais, escritos em uma linguagem (SAL) com suporte à programação estruturada. Patterson e Sequein (1982a) analisaram um conjunto de dados referentes à execução de compiladores, programas de formatação de textos, programas de CAD e procedimentos de ordenação e comparação de arquivos. As linguagens de programação C e Pascal foram também estudadas. Huck (1983) analisou quatro programas representativos de computação científica de propósito geral, incluindo a transformada rápida de Fourier e a integração de sistemas de equações diferenciais. Os resultados obtidos para todas essas linguagens e aplicações mostram que instruções de desvio e chamadas de sub-rotinas representam apenas uma pequena fração dos comandos executados por programas. A afirmação 1 é, portanto, confirmada por esses estudos.

A afirmação 2 é confirmada por estudos relatados por Patterson (1985a), mostrados na Figura 4.29, que apresenta o padrão de ocorrência de instruções de chamada e retorno de procedimentos. Uma chamada é representada por uma linha descendente e uma instrução de retorno, por uma linha ascendente. Nessa figura, uma janela é definida com um nível de profundidade igual a 5. Apenas uma seqüência de instruções de chamada e retorno de procedimentos de profundidade igual a 6 em qualquer direção causa uma movimentação da janela. Como se pode ver, a execução do programa permanece dentro de uma dada janela por longos períodos de tempo. Uma análise de programas C e Pascal mostrou que uma janela com nível de profundidade 8 é movimentada em menos de 1% das instruções de chamada ou retorno (Tamir, 1983).

O princípio de localidade de referências é também confirmado em estudos mais recentes. Por exemplo, a Figura 4.30 mostra os resultados de um estudo de padrões de acesso aos sites Web a partir de um determinado endereço.

A literatura geralmente distingue localidade espacial e localidade temporal. **Localidade espacial** refere-se à tendência de programas em fazer referências a posições de memória agrupadas. Isso reflete a tendência do processador de acessar instruções seqüencialmente. A localidade espacial reflete também a tendência de programas em acessar dados seqüencialmente, por exemplo, no processamento de uma tabela. **Localidade temporal** refere-se à tendência do processador em usar posições de memória utilizadas recentemente. Por exemplo, quando é executado um laço de uma iteração, o processador executa o mesmo conjunto de instruções repetidamente.

Tabela 4.9 Freqüência dinâmica relativa de operações em uma linguagem de alto nível

Estudo	(Huck, 1983)	(Knuth, 1971)	(Patterson, 1982)		(Tanenbaum, 1978)
Linguagem	Pascal	FORTRAN	Pascal	C	SAL
Natureza dos programas	Científica	Estudante	Sistema	Sistema	Sistema
Atribuição	74	67	45	38	42
Laço	4	3	5	3	4
Chamada de sub-rotina	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Outros	—	7	6	1	6

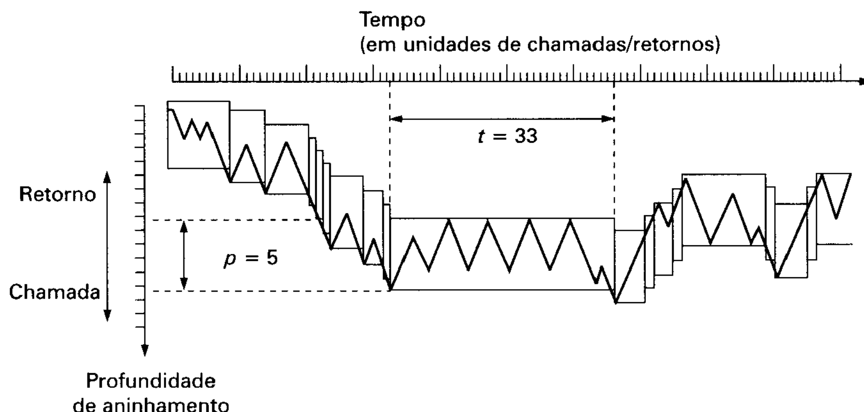


Figura 4.29 Comportamento de chamadas e retornos de procedimentos.

Operação de memória de dois níveis

A propriedade de localidade de referências pode ser explorada na construção de uma memória de dois níveis. A memória de nível superior (M1) é menor, mais rápida e mais cara

(custo por bit) do que a memória de nível inferior (M2). M1 é usada como uma área de armazenamento temporário para parte do conteúdo de M2. Quando ocorre uma referência à memória, procura-se obter em M1 o dado requerido. Se o dado estiver em M1, então o acesso é feito rapidamente. Caso contrário, um bloco de posições de memória é copiado de M2 para M1, ocorrendo então o acesso, via M1. Em função da localidade de referências, uma vez que um bloco é trazido para M1, deve ocorrer certo número de referências a dados nesse bloco, resultando de maneira geral em acessos rápidos à memória.

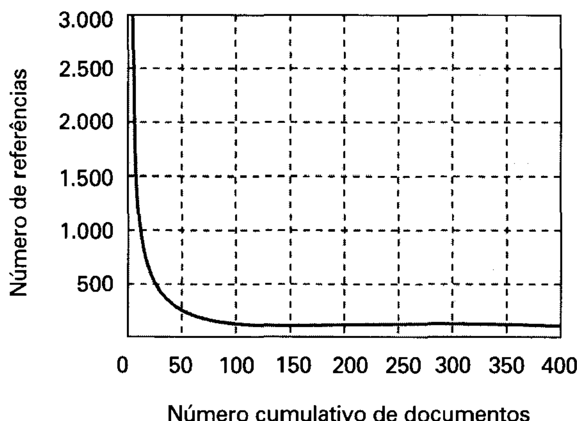


Figura 4.30 Localidade de referências a páginas Web (Baentsch, 1997).

Para expressar o tempo médio de acesso a um dado, devemos considerar não apenas as velocidades dos dois níveis de memória, mas também a probabilidade de que uma dada referência seja encontrada em M1. Temos:

$$T_s = H \times T_1 + (1 - H) \times (T_1 + T_2)$$
$$= T_1 + (1 - H) \times T_2 \quad (4.1)$$

onde:

T_s = tempo médio de acesso (sistema)
T_1 = tempo de acesso a M1 (por exemplo, cache, cache de disco)
T_2 = tempo de acesso a M2 (por exemplo, memória principal, disco)
H = taxa de acerto (fração das referências encontradas em M1)

A Figura 4.2 mostra o tempo médio de acesso em função da taxa de acerto. Como se pode ver, para uma porcentagem alta de acertos, o tempo de acesso total médio é muito mais próximo do tempo de acesso de M1 do que de M2.

Desempenho

Examinamos agora alguns parâmetros relevantes no projeto de uma memória de dois níveis. Consideramos, primeiramente, o preço:

$$C_S = \frac{C_1S_1 + C_2S_2}{S_1 + S_2} \quad (4.2)$$

onde:

C_S = custo médio por bit para a memória combinada de dois níveis

C_1 = custo médio por bit da memória de nível superior M1

C_2 = custo médio por bit da memória de nível inferior M2

S_1 = tamanho de M1

S_2 = tamanho de M2

Gostaríamos de ter $C_s \approx C_2$. Dado que $C_1 >> C_2$, isso requer $S_1 << S_2$. A Figura 4.31 mostra essa relação.

Consideramos agora o tempo de acesso. Para que uma memória de dois níveis apresente um aumento significativo de desempenho, devemos ter T_s aproximadamente igual a T_1 ($T_s \approx T_1$). Como T_1 é muito menor que T_2 ($T_1 << T_2$), devemos ter uma taxa de acerto próxima de 1.

Queremos então que M1 seja menor, para diminuir o custo, e que seja grande, para aumentar a taxa de acerto e, conseqüentemente, melhorar o desempenho. Existirá um tamanho de M1 que satisfaça esses dois requisitos de maneira razoável? Essa questão pode ser respondida por meio das respostas a uma série de questões relacionadas:

- Qual é o valor da taxa de acerto requerido para se obter o desempenho desejado?
- Que tamanho de M1 asseguraria essa taxa de acerto?
- Esse tamanho atende aos requisitos de custo?

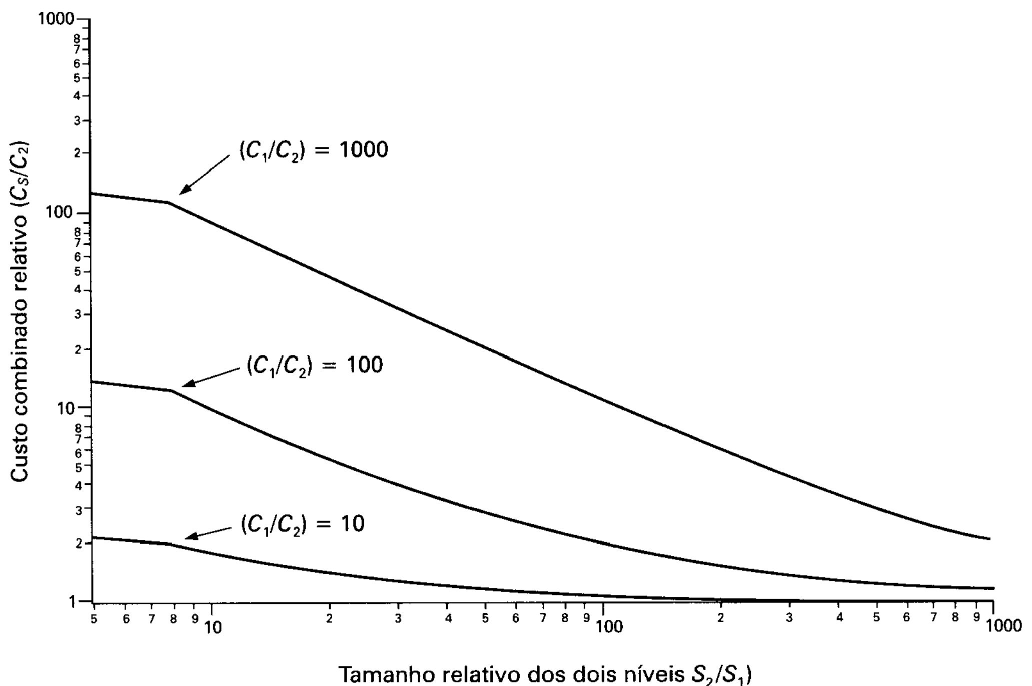


Figura 4.31 Relação entre custo médio e tamanho relativo em uma memória de dois níveis.

Para responder a essas questões, considere o valor T_1 / T_s, conhecido como eficiência de acesso. Ele indica quão próximo o tempo médio de acesso (T_s) está do tempo de acesso de M1 (T_1). Da Equação 4.1, obtemos:

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\frac{T_2}{T_1}} \tag{4.3}$$

Na Figura 4.32, mostramos o gráfico de T_1 / T_s em função da taxa de acerto H, tendo T_2 / T_1 como parâmetro. Tipicamente, o tempo de acesso à memória cache é cinco a dez vezes mais rápido do que o tempo de acesso à memória principal (isto é, T_2 / T_1 varia de 5 a 10) e o tempo de acesso à memória principal é cerca de mil vezes mais rápido do que o tempo de acesso ao disco (T_2 / T_1 = 1.000). Dessa maneira, parece ser necessária uma taxa de acerto entre 0,8 e 0,9 para satisfazer os requisitos de desempenho.

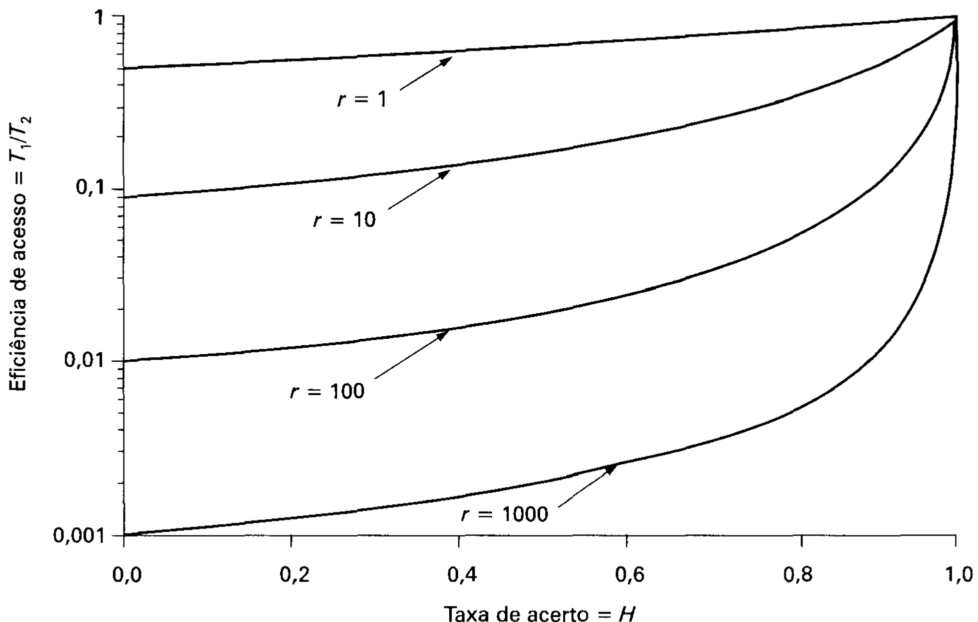


Figura 4.32 Eficiência de acesso como função da taxa de acerto ($r = T_2 / T_1$).

Podemos agora expressar mais precisamente a questão do tamanho relativo da memória. Uma taxa de acerto de 0,8 ou maior é razoável para $S_1 << S_2$? Isso vai depender de diversos fatores, incluindo a natureza do software a ser executado e os detalhes do projeto da memória de dois níveis. O aspecto mais importante é, naturalmente, o grau de localidade de referências. A Figura 4.33 sugere o efeito da localidade de referências sobre a taxa de acerto. Claramente, se M1 tem o mesmo tamanho que M2, a taxa de acerto é de 1,0: todos os dados de M2 estão também armazenados em M1. Suponha agora que não exista localidade de referências, isto é, que as referências são completamente aleatórias. Nesse caso, a taxa de acerto é

uma função estritamente linear do tamanho relativo da memória. Por exemplo, se M1 tem metade do tamanho de M2, a qualquer instante a metade dos dados de M2 está também em M1 e a taxa de acerto é de 0,5. Na prática, existe sempre algum grau de localidade de referências. Os efeitos de uma localidade de referências alta ou moderada são indicados na figura.

Portanto, se existe uma forte localidade de referências, é possível obter uma taxa de acerto bastante alta, mesmo com um tamanho relativamente pequeno da memória de nível superior. Numerosos estudos mostram que memórias cache de tamanho bastante pequeno obtém uma taxa de acerto acima de 0,75, *independentemente do tamanho da memória principal* (por exemplo, Agarwal, 1989a, Przybylski, 1988, Strecker, 1983, e Smith, 1982). Uma memória cache de 1K a 128K palavras geralmente é adequada, considerando que o tamanho típico da memória principal é, atualmente, da ordem de vários megabytes. Quando considerarmos o sistema de memória virtual e a memória cache de disco, veremos outros estudos que confirmam o mesmo fenômeno — ou seja, que mesmo uma memória M1 de tamanho relativamente pequeno obtém uma alta taxa de acerto, em razão da localidade das referências à memória.

Isso nos leva à última questão relacionada anteriormente: O tamanho relativo das duas memórias satisfaz os requisitos de custo? A resposta é, claramente, sim. Se precisamos apenas de uma memória de nível superior relativamente pequena para obter um bom desempenho, então o custo médio por bit dos dois níveis de memória se aproxima do custo por bit na memória de nível inferior, que é mais barata.

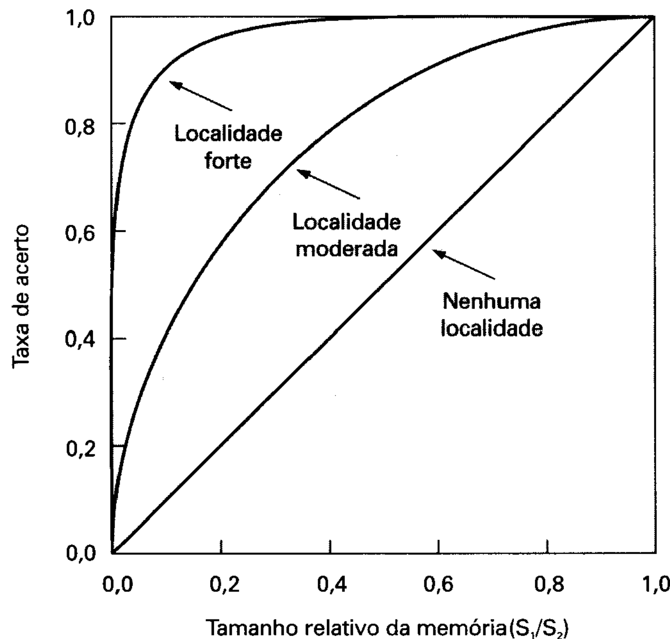
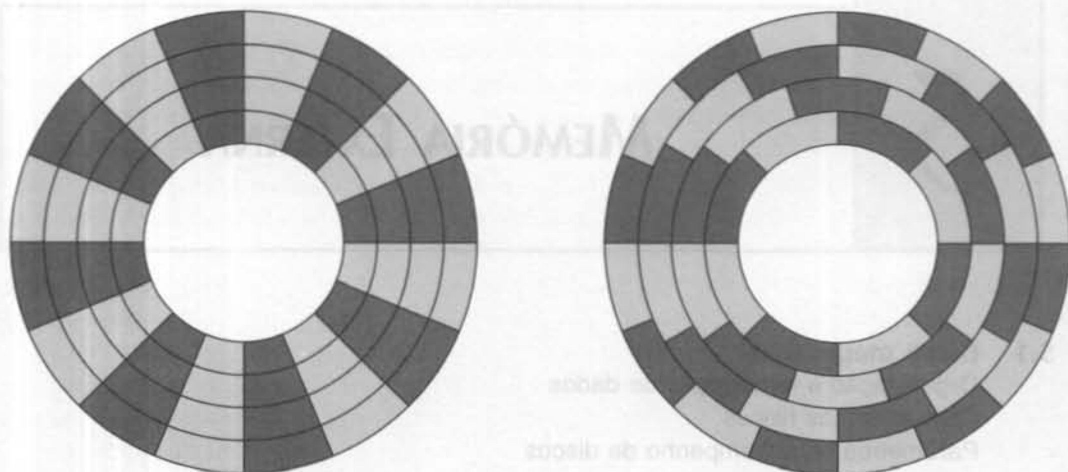


Figura 4.33 Taxa de acerto em função do tamanho relativo da memória.

capítulo

5 Memória Externa



- Os discos magnéticos ainda constituem o componente mais importante da memória externa. Os discos removíveis e os discos fixos, ou rígidos, são usados em sistemas de computação que vão desde os computadores pessoais até os supercomputadores.
- A tecnologia de discos RAID é bastante usada para aumentar o desempenho e a disponibilidade dos dados em servidores e sistemas de grande porte. A sigla RAID refere-se a uma família de técnicas para a utilização de vários discos agrupados em paralelo, empregando redundância para compensar eventuais falhas nos discos.
- A tecnologia de armazenamento óptico vem se tornando cada vez mais importante em todos os tipos de sistemas de computação. Embora o CD-ROM já venha sendo largamente usado há vários anos, outras tecnologias mais recentes, como o CD regravável e os dispositivos de armazenamento magneto-ópticos, estão se tornando cada vez mais importantes.

Este capítulo aborda uma variedade de dispositivos e sistemas de memória externa. Começamos pelo dispositivo mais importante, o disco magnético. Os discos magnéticos constituem a base da memória externa de quase todos os sistemas de computação. Em seguida, discutimos o uso de vários discos para obtenção de maior desempenho, tratando especificamente da família de sistemas conhecida como RAID (*redundant array of independent disks* — agrupamento redundante de discos independentes). Um componente de importância crescente em muitos sistemas de computação é a memória externa óptica, que é abordada na terceira seção. A seção final trata das fitas magnéticas.

5.1 DISCO MAGNÉTICO

O disco magnético é constituído de um prato circular de metal ou de plástico, coberto com um material que pode ser magnetizado. Os dados são gravados e posteriormente lidos do disco por meio de uma bobina condutora denominada **cabeçote**, conhecida também como cabeça de leitura/gravação. Durante uma operação de escrita ou de leitura, o cabeçote permanece estático, enquanto o prato gira embaixo dele.

O mecanismo de escrita é baseado no fato de que o fluxo de corrente elétrica por meio de uma bobina produz um campo magnético. São enviados pulsos de corrente para o cabeçote, que resultam na gravação de padrões magnéticos na superfície abaixo dele; correntes positivas e negativas geram padrões magnéticos distintos. O mecanismo de leitura é baseado no fato de que um campo magnético que se move em relação a uma bobina produz uma corrente elétrica nessa bobina. Quando a superfície do disco passa sob o cabeçote, ela gera uma corrente de polaridade igual à da corrente utilizada na gravação.

Organização e formatação de dados

O cabeçote é um dispositivo relativamente pequeno, capaz de ler ou escrever sobre uma região do prato que gira embaixo dele. Isso resulta em uma organização dos dados (também conhecida como leiaute de dados do disco) no prato em forma de anéis concêntricos, denominados **trilhas**. Cada trilha tem a mesma largura do cabeçote. Existem, tipicamente, 500 a 2000 trilhas por superfície.

A Figura 5.1 representa essa organização dos dados. Trilhas adjacentes são separadas por **espaços** (*gaps*). Isso evita, ou pelo menos diminui, a ocorrência de erros devidos à falta de alinhamento do cabeçote ou à interferência de campos magnéticos. Para simplificar o circuito eletrônico envolvido, um mesmo número de bits é armazenado em cada trilha. Dessa maneira, a **densidade**, em número de bits por centímetro linear, aumenta a partir da trilha mais externa para a mais interna.

Os dados são transferidos do e para o disco em **blocos**. Normalmente, um bloco tem tamanho menor do que a capacidade de uma trilha. Os dados são armazenados em regiões do mesmo tamanho de um bloco, denominadas **setores** (Figura 5.1). Existem, tipicamente, entre 10 e 100 setores por trilha, que podem ter tamanho fixo ou variável. Para evitar impor requisitos de precisão exagerados ao sistema, setores adjacentes são separados por espaços internos à trilha (ou espaços entre setores).

O sistema requer a existência de algum mecanismo para localizar um setor dentro de uma trilha. Naturalmente, devem existir um ponto de início da trilha e uma maneira de identificar o início e o fim de cada setor. Essas informações são fornecidas por dados de controle gravados no disco. Desse modo, um disco é formatado com alguns dados extras, usados apenas pela unidade de disco e são invisíveis para o usuário.

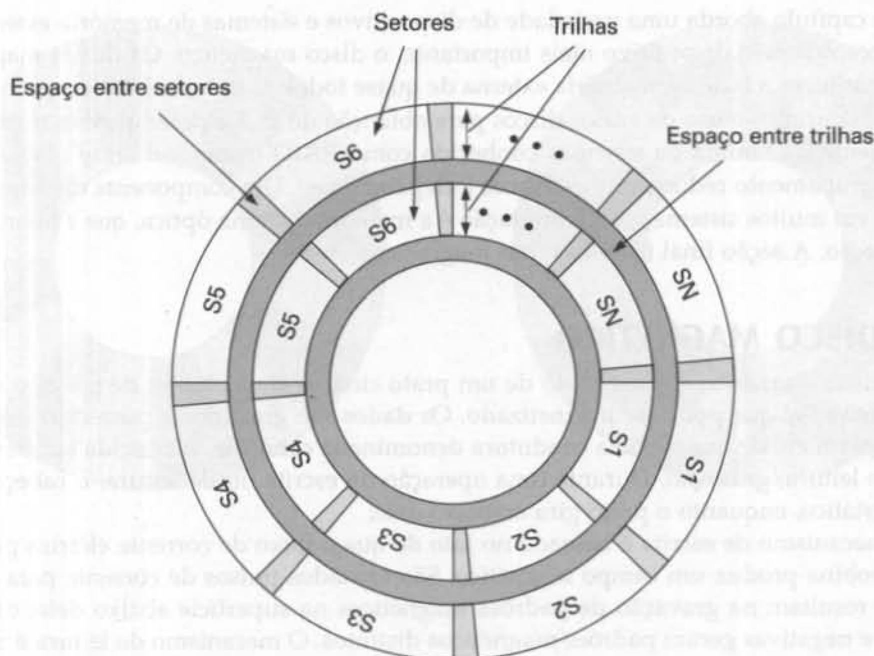


Figura 5.1 Organização dos dados no disco.

Um exemplo de formatação de um disco é mostrado na Figura 5.2. Nesse caso, cada trilha contém 30 setores, com tamanho fixo de 600 bytes cada. Cada setor contém 512 bytes de dados e informações de controle para o controlador de disco. O campo ID consiste em um identificador ou endereço utilizado para localizar um determinado setor. O byte SYNCH contém um padrão de bits especial que indica o início de um campo. O número de trilha identifica uma trilha na superfície e o número de cabeçote, um cabeçote, uma vez que esse disco tem múltiplas superfícies. Os campos de ID e de dados contêm um código de correção de erros.

Características físicas

A Tabela 5.1 (página 168) enumera as características mais importantes que diferenciam os diversos tipos de discos magnéticos. Primeiramente, o cabeçote pode ser fixo ou móvel em relação à direção radial do prato. Em um **disco de cabeçote fixo** existe um cabeçote de leitura e escrita para cada trilha. Os cabeçotes são montados em um braço rígido que se estende por todas as trilhas (Figura 5.3a). Em um **disco de cabeçote móvel** existe apenas um cabeçote de leitura e escrita (Figura 5.3b). Esse cabeçote é também montado em um braço. Como o cabeçote deve poder ser posicionado sobre qualquer trilha, o braço pode ser estendido ou retraído.

O disco propriamente dito é montado em uma unidade de disco, que consiste em um braço, um eixo para girar o disco e circuitos eletrônicos para a entrada e a saída de dados binários. Um **disco não-removível** é montado permanentemente na unidade de disco. Um **disco removível** pode ser removido e substituído por outro disco. A vantagem dos discos removíveis é possibilitar o armazenamento de uma quantidade ilimitada de dados usando um número limitado de unidades de disco e um número ilimitado de discos. Além disso, esse disco pode ser transportado de um computador para outro.

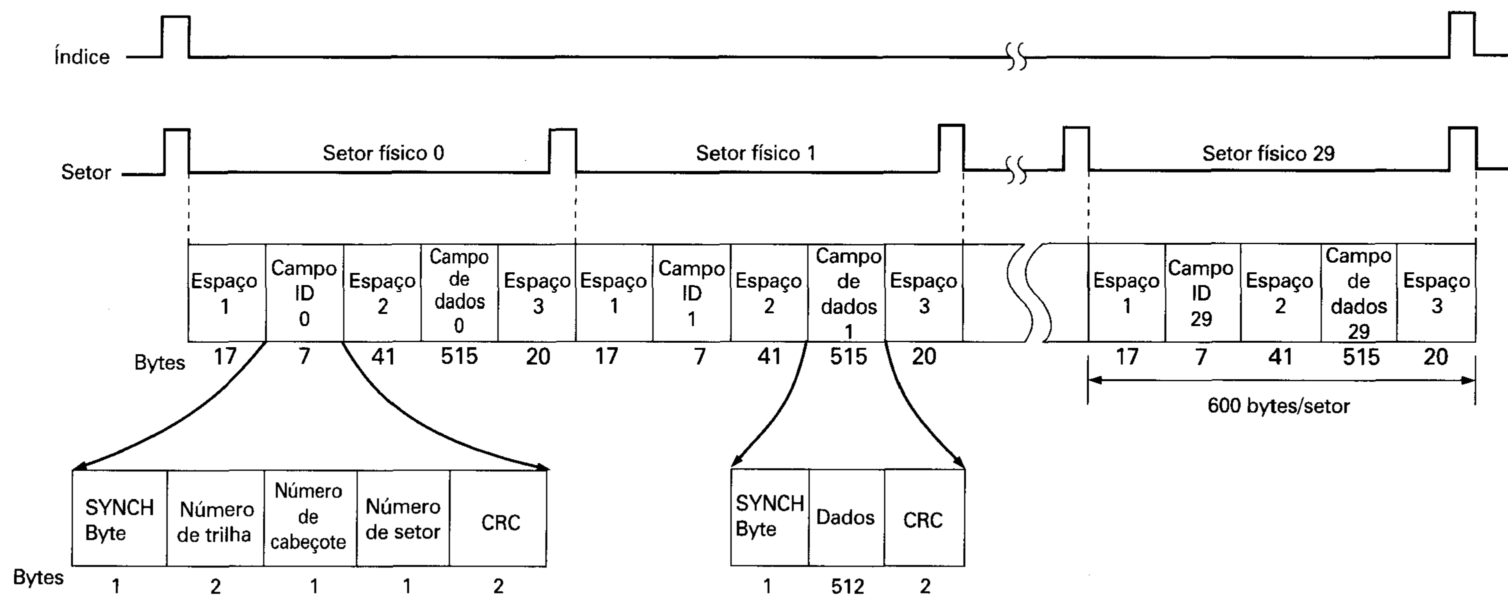


Figura 5.2 Formato de trilha de disco Winchester (Seagate ST506).

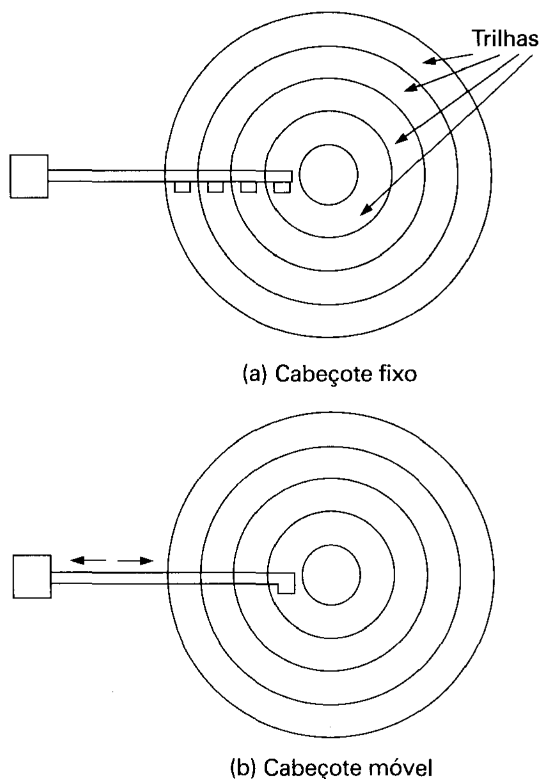


Figura 5.3 Discos com cabeçotes fixos e móveis.

Na maioria dos discos, a cobertura magnetizável é aplicada nos dois lados do prato, sendo assim denominados de **duplo lado**. Alguns sistemas de disco mais baratos usam discos de um **único lado**.

Algumas unidades de disco acomodam **múltiplos pratos**, empilhados verticalmente e separados por cerca de 2,5 cm (Figura 5.4). São usados braços múltiplos. Os pratos são agrupados em unidades denominadas **pacotes de disco**.

Os discos são classificados em três tipos de acordo com o mecanismo de cabeçote utilizado. Tradicionalmente, o cabeçote de leitura e escrita é posicionado a uma distância fixa acima do prato, existindo uma fina camada de ar entre o cabeçote e o prato. Outro mecanismo utiliza cabeçotes que de fato tocam o prato durante uma operação de leitura ou escrita. Esse tipo de mecanismo é utilizado no **disco flexível** (ou disquete), que consiste em um prato pequeno e flexível e é o tipo mais barato de disco.

Para entender o terceiro tipo de disco, é necessário conhecer a relação entre a densidade dos dados e a altura da camada de ar entre o cabeçote e o prato. Para ler ou escrever um dado, o cabeçote tem de gerar ou sentir um campo eletromagnético de magnitude suficiente. Quanto mais estreito é o cabeçote, mais próximo ele deve estar da superfície do prato para funcionar corretamente. Um cabeçote mais estreito possibilita trilhas mais estreitas e, portanto, maior densidade de dados, o que é desejável. Entretanto, quanto mais próximo o cabeçote estiver do disco, maior será o risco de ocorrência de erros devidos a impurezas ou imperfeições. Um avanço na tecnologia de discos foi obtido com o desenvolvimento dos discos Winchester. Os

cabeçotes do Winchester são montados em unidades de disco lacradas, quase livres de contaminações. São projetados para operar mais perto da superfície do disco do que os cabeçotes dos discos rígidos convencionais, possibilitando assim maior densidade de dados. São, de fato, uma lâmina aerodinâmica que descansa levemente sobre a superfície do prato quando o disco está imóvel. A pressão do ar gerada pelo disco ao girar é suficiente para fazer a lâmina erguer-se acima da superfície. O sistema sem contato resultante pode ser projetado para usar cabeçotes mais estreitos, que podem operar mais próximos da superfície do prato do que os cabeçotes dos discos rígidos convencionais[1].

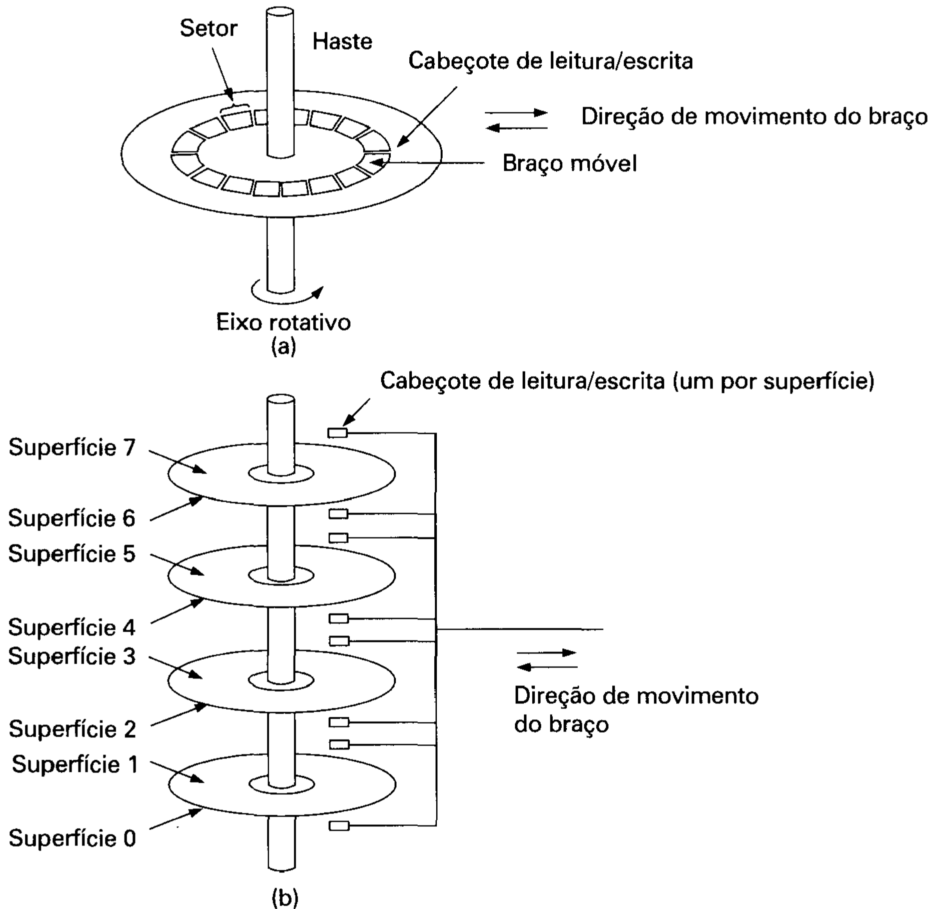


Figura 5.4 Disco de múltiplos pratos.

1 O termo *Winchester* foi originalmente usado pela IBM como um codinome para o modelo de disco 3340, antes do seu lançamento no mercado. O 3340 era composto de um pacote de discos removíveis, com cabeçotes lacrados dentro da cápsula. O termo é empregado atualmente para qualquer unidade de discos lacrada, que utiliza cabeçote aerodinâmico. O disco Winchester é usado tanto em computadores pessoais quanto em estações de trabalho, onde é conhecido como *disco rígido*.

Tabela 5.1 Características físicas de sistemas de disco

Movimento do cabeçote	**Pratos**
Cabeçote fixo (um por trilha)	Prato único
Cabeçote móvel (um por superfície)	Múltiplos pratos
Transportabilidade do disco	**Mecanismo do cabeçote**
Disco não-removível	Contato (disquete)
Disco removível	Espaço fixo
Lados	Espaço aerodinâmico (Winchester)
Lado único	
Duplo lado	

Parâmetros de desempenho de discos

Os detalhes de uma operação de E/S em um disco dependem do sistema de computação, do sistema operacional e do hardware usado no canal de E/S e no controlador de disco. O diagrama de tempo de uma operação de E/S em um disco é mostrado na Figura 5.5.

Quando uma unidade de disco está em operação, o disco gira a uma velocidade constante. Para ler ou escrever um valor, o cabeçote deve ser posicionado sobre a trilha desejada e no início do setor desejado da trilha. A seleção da trilha requer a movimentação do cabeçote, em um sistema de cabeçote móvel, ou a seleção eletrônica de um dos cabeçotes, em um sistema de cabeçote fixo. Em um sistema de cabeçote móvel, o tempo para posicionar o cabeçote na trilha é denominado **tempo de busca** (*seek time*). Em ambos os sistemas, uma vez selecionada a trilha, o controlador de disco espera que o disco gire até que o setor desejado esteja alinhado com o cabeçote. O tempo decorrido até que o início do setor esteja sob o cabeçote é denominado **atraso rotacional** ou latência rotacional. A soma do tempo de busca, se houver, com o atraso rotacional é denominada **tempo de acesso**, isto é, o tempo requerido para atingir a posição em que deve ser feita a leitura ou a escrita. Uma vez que o cabeçote esteja na posição correta, a operação de leitura ou escrita é feita à medida que o setor se move sob o cabeçote; essa parte da operação corresponde à transferência de dados.

Além do tempo de acesso e do tempo de transferência, existem normalmente vários atrasos associados a uma operação de E/S em um disco. Quando um processo faz uma requisição de E/S, ela deve primeiro esperar em uma fila até que o dispositivo esteja disponível. O dispositivo é então alocado para o processo. Se o dispositivo compartilha um canal de E/S ou um conjunto de canais de E/S com outras unidades de disco, pode haver uma espera adicional para que o canal fique disponível. Quando isso ocorre, a operação de busca da trilha pode ser iniciada.

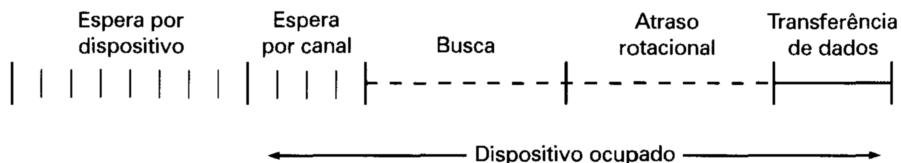


Figura 5.5 Tempo de uma operação de E/S em um disco.

Em alguns sistemas de grande porte, é usada uma técnica conhecida como detecção de posição de rotação (*rotational positional sensing* — RPS). Essa técnica funciona da maneira des-

crita a seguir: quando uma operação de busca no disco é iniciada, o canal é liberado para atender a outras operações de E/S. Ao finalizar a busca, o dispositivo determina quando os dados estarão posicionados sob o cabeçote. Assim que o setor requerido se aproxima do cabeçote, o dispositivo tenta restabelecer a comunicação com o sistema. Se a unidade de controle ou o canal estiverem ocupados com outra operação de E/S, a tentativa de conexão falhará e o dispositivo terá de girar um ciclo inteiro antes de tentar uma nova conexão; isso é denominado uma perda de RPS. Esse atraso extra deve ser acrescentado ao diagrama de tempo da Figura 5.5.

Tempo de busca

O tempo de busca é o tempo necessário para mover o braço do disco até a trilha desejada. Ele é difícil de ser medido. Constitui-se de dois componentes principais: o tempo inicial de partida e o tempo requerido para percorrer as trilhas depois que o braço de acesso está pronto para se movimentar. Esse tempo de percurso, infelizmente, não é uma função linear do número de trilhas percorridas. O tempo de busca pode ser aproximado pela seguinte fórmula linear:

$$T_s = m \times n + s$$

onde:

T_s = tempo estimado de busca
m = constante que depende da unidade de disco
n = número de trilhas percorridas
s = tempo de partida

Por exemplo, em um disco rígido barato de um computador pessoal, os valores desses parâmetros são, aproximadamente, m = 0,3 ms e s = 20 ms. Em um disco maior e mais caro, podemos ter m = 0,1 ms e s = 3 ms.

Atraso rotacional

Discos giram tipicamente a 3600 rpm* (exceto os discos flexíveis), ou seja, efetuam uma revolução a cada 16,7 ms. Dessa maneira, o atraso rotacional é, em média, de 8,3 ms. Discos flexíveis giram tipicamente a velocidades entre 300 e 600 rpm. Assim, o atraso rotacional médio é de 100 a 200 ms.

Tempo de transferência

O tempo de transferência do disco depende da sua velocidade de rotação, conforme a seguinte relação:

$$T = \frac{b}{rN}$$

onde:

T = tempo de transferência
b = números de bytes transferidos
N = número de bytes na trilha
r = velocidade de rotação em número de revoluções por segundo

* N.R.T.: Discos modernos têm uma rotação típica de 7200 rpm. Existem discos com rotação superior a 10.000 rpm.

Portanto, o tempo total de acesso médio pode ser expresso como:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

onde T_S é o tempo médio de busca.

Comparação de tempos de acesso

Uma vez definidos os parâmetros anteriores, examinamos duas diferentes operações de E/S que mostram o perigo de se confiar em valores médios. Considere um disco típico com tempo médio de busca anunciado pelo fabricante de 20 ms, taxa de transferência de 1 Mbyte/s e setores de 512 bytes, com 32 setores por trilha. Suponha que desejamos ler um arquivo de 128 Kbytes com 256 setores. Estimamos o tempo total para a transferência.

Primeiramente, supomos que o arquivo está armazenado no disco da maneira mais compacta possível. Ou seja, o arquivo ocupa todos os setores de oito trilhas adjacentes (8 trilhas × 32 setores/trilhas = 256 setores). Essa organização é conhecida como *organização seqüencial*. O tempo para a leitura da primeira trilha é calculado do seguinte modo:

Tempo médio de busca	20,0	ms
Atraso rotacional	8,3	ms
Leitura de 32 setores	16,7	ms
	45	ms

Suponha que as trilhas restantes possam ser lidas essencialmente sem tempo de busca. Ou seja, a operação de E/S pode acompanhar a velocidade de fluxo do disco. Precisamos então lidar apenas com o atraso rotacional para cada trilha subseqüente. Dessa maneira, cada trilha sucessiva é lida em 8,3 + 16,7 = 25 ms. Para ler o arquivo inteiro, temos:

Tempo total = 45 + 7 × 25 = 220 ms = 0,22 s

Agora calculamos o tempo necessário para ler os mesmos dados, usando acesso aleatório em vez de acesso seqüencial, isto é, supondo que os setores são distribuídos aleatoriamente pelo disco. Para cada setor, temos:

Tempo médio de busca	20,0	ms
Atraso rotacional	8,3	ms
Leitura de um setor	0,5	ms
	28,8	ms

Tempo total = 256 × 28,8 = 7373 ms = 7,37 s

É claro que a ordem em que os setores são obtidos no disco tem efeito significativo sobre o desempenho de E/S. No caso de acessos a arquivos em que a leitura ou a escrita é feita em múltiplos setores, é possível ter algum controle sobre o modo como os setores de dados são organizados no disco, como veremos no decorrer do próximo capítulo. Entretanto, em um ambiente de multiprogramação, mesmo no caso de acesso a um único arquivo, existirão diversas requisições de E/S competindo por um mesmo disco. Dessa maneira, vale a pena examinar como o desempenho de operações de E/S pode ser melhorado em relação ao desempenho obtido quando o acesso ao disco é puramente aleatório. Essas considerações le-

vam à utilização de algoritmos de escalonamento do disco, assunto que foge ao escopo deste livro, uma vez que é um tópico relativo ao sistema operacional (veja Stallings, 1998, para uma discussão sobre o assunto).

5.2 RAID

Como foi discutido anteriormente, a melhoria no desempenho de memórias secundárias tem sido consideravelmente menor do que a de processadores e da memória principal. Essa diferença fez dos sistemas de armazenamento em discos o principal foco de preocupação para melhoria do desempenho global de sistemas de computação.

Assim como em outras áreas, os projetistas de armazenamento de disco sabem que, se um dispositivo pode ser melhorado apenas até certo ponto, ganhos adicionais de desempenho podem ser obtidos utilizando vários componentes em paralelo. No caso de armazenamento de disco, essa idéia levou ao desenvolvimento de um agrupamento de discos que operam independentemente e em paralelo. Com diversos discos, diferentes requisições de E/S podem ser processadas em paralelo, desde que os dados requeridos residam em discos separados. Mais do que isso, uma única requisição de E/S poderá também ser executada em paralelo, se o bloco de dados a ser acessado for distribuído em vários discos.

Com o uso de múltiplos discos, os dados podem ser organizados de diversas maneiras, podendo ser empregada alguma redundância para melhorar a confiabilidade. A possibilidade de organizar os dados de vários modos poderia tornar difícil o desenvolvimento de bancos de dados compatíveis com diferentes plataformas e sistemas operacionais. Felizmente, a indústria decidiu adotar um padrão para o projeto de bancos de dados de vários discos, conhecido como RAID (agrupamento redundante de discos independentes). O esquema RAID consiste em sete níveis[2], de zero a seis. Esses níveis não implicam em uma relação hierárquica, mas designam diferentes arquiteturas de projeto que compartilham três características comuns:

1. O RAID consiste em um agrupamento de unidades de discos físicos, visto pelo sistema operacional como uma única unidade de disco lógico.
2. Os dados são distribuídos pelas unidades de discos físicos do agrupamento.
3. A capacidade de armazenamento redundante é utilizada para armazenar informação de paridade, garantindo a recuperação dos dados em caso de falha em algum disco.

Os detalhes da segunda e da terceira características diferem para os diferentes níveis de RAID. O RAID 0 não oferece a terceira característica.

O termo *RAID* foi originalmente empregado em um artigo de um grupo de pesquisadores da Universidade da Califórnia, em Berkeley (Patterson, 1988)[3]. Esse artigo esboçava várias configurações e aplicações de RAID e introduzia as definições dos níveis de RAID usados até hoje. A estratégia RAID substitui as unidades de disco de grande capacidade por várias uni-

---

2 Níveis adicionais de RAID foram definidos por alguns pesquisadores e algumas companhias, mas os sete níveis descritos nesta seção são os únicos aceitos universalmente.

3 Nesse artigo, o acrônimo RAID significava agrupamento redundante de discos de baixo custo. O termo *baixo custo* foi empregado para contrapor os discos pequenos e relativamente baratos do agrupamento RAID à tecnologia de um único disco grande e caro (*single large expensive disk* — sled). O sled tornou-se, essencialmente, coisa do passado; a mesma tecnologia é hoje usada, em configurações RAID ou em outras configurações. Assim, a indústria passou a adotar o termo *independente* para enfatizar que o agrupamento RAID fornece ganhos significativos de desempenho e confiabilidade.

dades de capacidade menor, distribuindo os dados para possibilitar acessos simultâneos nas várias unidades e, desse modo, melhorar o desempenho de E/S e facilitar o aumento significativo de capacidade da memória secundária.

Uma característica única da tecnologia RAID se refere ao uso eficaz de redundância de dados. Embora o uso simultâneo de vários cabeçotes e discos possibilite obter taxas de transferência de E/S mais altas, isso aumenta também a probabilidade de falhas. Para compensar essa diminuição de confiabilidade, o RAID usa a informação de paridade armazenada que possibilita a recuperação dos dados perdidos devido a uma falha de disco.

As características de cada um dos níveis de RAID são examinadas a seguir. A Tabela 5.2 resume as características dos sete níveis. Os níveis 2 e 4 ainda não estão disponíveis no mercado e existe pequena probabilidade de serem aceitos industrialmente. Entretanto, a descrição desses níveis auxilia na compreensão de decisões de projeto em alguns dos demais níveis.

Tabela 5.2 Níveis de RAID

Categoria	Nível	Descrição	Taxa de requisição de E/S (leitura/ escrita)	Taxa de transferência de dados (leitura/ escrita)	Aplicação típica
Intercalação de dados (*striping*)	0	Não-redundante	Tiras grandes: excelente	Tiras pequenas: excelente	Aplicações que requerem alto desempenho para dados não-críticos
Espelhamento	1	Espelhado	Bom/ razoável	Razoável/ razoável	Unidades de disco de sistema; arquivos críticos
Acesso paralelo	2	Redundante via código de Hamming	Pobre	Excelente	
	3	Paridade de bit intercalada	Pobre	Excelente	Aplicações com grandes requisições de E/S, como processamento de imagens e CAD
Acesso independente	4	Paridade de bloco intercalada	Excelente/ razoável	Razoável/ pobre	
	5	Paridade de bloco intercalada e distribuída	Excelente/ razoável	Razoável/ pobre	Buscas de dados, com altas taxas de requisição e grande volume de leituras
	6	Paridade de bloco dupla intercalada e distribuída	Excelente/ pobre	Razoável/ pobre	Aplicações que requerem grande disponibilidade de dados

A Figura 5.6 mostra um exemplo de uso dos sete esquemas RAID para implementar uma capacidade de dados que requer quatro discos, excluindo a redundância. A figura enfatiza a organização dos dados de usuário e dos dados redundantes e indica os requisitos de armazenamento dos vários níveis. Essa figura é utilizada como base para a discussão a seguir.

RAID de nível 0

O RAID de nível 0 não constitui de fato um membro da família RAID, uma vez que não inclui redundância para a melhora do desempenho. Contudo, ele é utilizado em poucas aplicações, como em supercomputadores, nos quais o desempenho e a capacidade constituem requisitos primordiais e o baixo custo é mais importante do que maior confiabilidade.

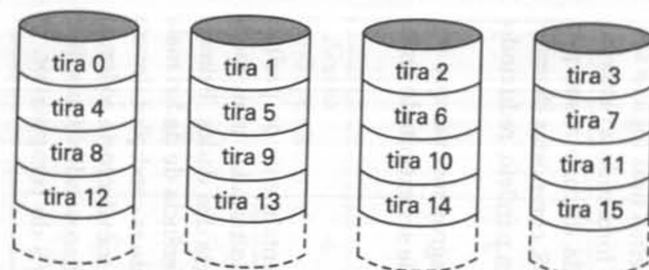
No RAID 0, os dados de usuário e de sistema são distribuídos em todos os discos do agrupamento. Essa distribuição dos dados tem enorme vantagem em relação ao uso de um único disco grande: se existirem duas requisições de E/S pendentes para dois blocos de dados distintos, haverá grande probabilidade de que esses blocos estejam em discos diferentes. Portanto, as duas requisições podem ser atendidas em paralelo, reduzindo o tempo de espera na fila de E/S.

Assim como os demais níveis de RAID, o RAID 0 não se limita a distribuir os dados pelo agrupamento de discos. Os dados são *intercalados em tiras* (*strips*) por meio dos discos disponíveis. Isso pode ser bem compreendido observando-se a Figura 5.7. Os dados de usuário e de sistema são vistos como sendo armazenados em um disco lógico. Esse disco é dividido em tiras; as tiras podem ser constituídas de blocos físicos, setores ou alguma outra unidade de armazenamento. As tiras são mapeadas sobre membros consecutivos do agrupamento de forma circular, cujo modo de distribuição é mais conhecido como *round robin*. Um conjunto de tiras logicamente consecutivas, cada qual mapeada em um dos membros do agrupamento, é denominada *faixa* (*stripe*). Em um agrupamento de n discos, as n primeiras tiras lógicas são fisicamente armazenadas como a primeira tira de cada um dos n discos, formando a primeira faixa; as n tiras seguintes são distribuídas como a segunda tira de cada disco, e assim por diante. A vantagem desse esquema é que, em grandes requisições de E/S, compostas de múltiplas tiras logicamente contíguas, até n tiras podem ser manipuladas em paralelo, reduzindo bastante o tempo de transferência de E/S.

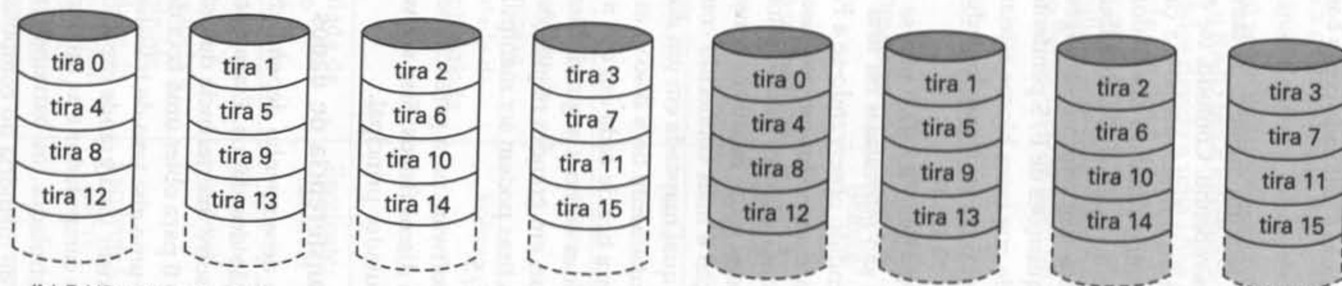
A Figura 5.7 indica o uso de um software para gerenciamento do agrupamento, que faz o mapeamento entre o disco lógico e os discos físicos. Esse software pode ser executado tanto no subsistema de disco como no computador principal.

RAID 0 para alta capacidade de transferência de dados

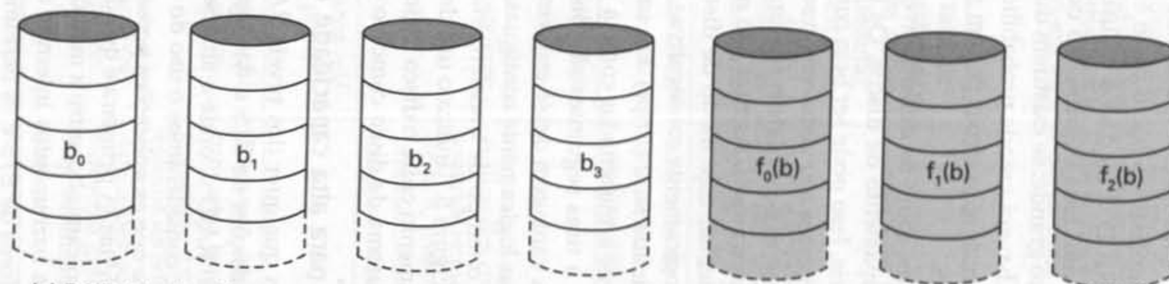
Em qualquer dos níveis RAID, o desempenho depende fundamentalmente do padrão das requisições de E/S e da organização dos dados. Essas questões são tratadas mais facilmente no RAID 0, onde a análise não sofre interferência da redundância dos dados. Primeiramente, consideramos o uso do RAID 0 para obter uma taxa de transferência de dados mais alta. Para que as aplicações possam ter uma alta taxa de transferência, duas condições devem ser satisfeitas. A primeira é que deve existir uma grande capacidade de transferência ao longo de todo o caminho entre a memória do computador e as unidades de discos individuais. Isso inclui os barramentos internos do controlador, os barramentos de E/S do computador, os adaptadores de E/S e os barramentos de memória do computador.



a) RAID 0 (não-redundante)

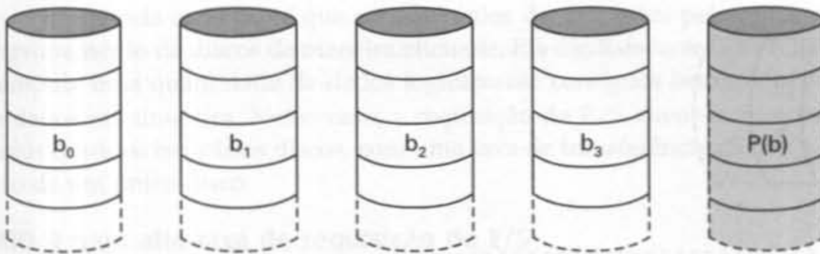


(b) RAID 1 (espelhado)

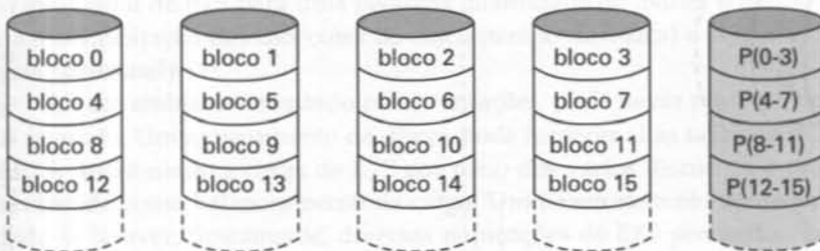


(c) RAID 2 (redundância via código de Hamming)

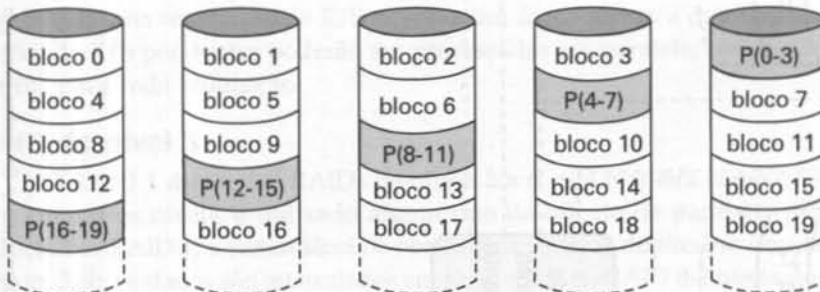
Figura 5.6 Níveis de RAID.



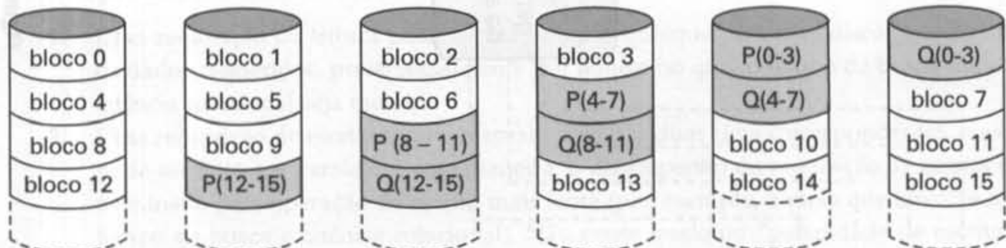
(d) RAID 3 (bit de paridade intercalado)



(e) RAID 4 (paridade de bloco)



(f) RAID 5 (paridade de bloco distribuída)



(g) RAID 6 (redundância dupla)

Figura 5.6 Níveis de RAID. (*continuação*)

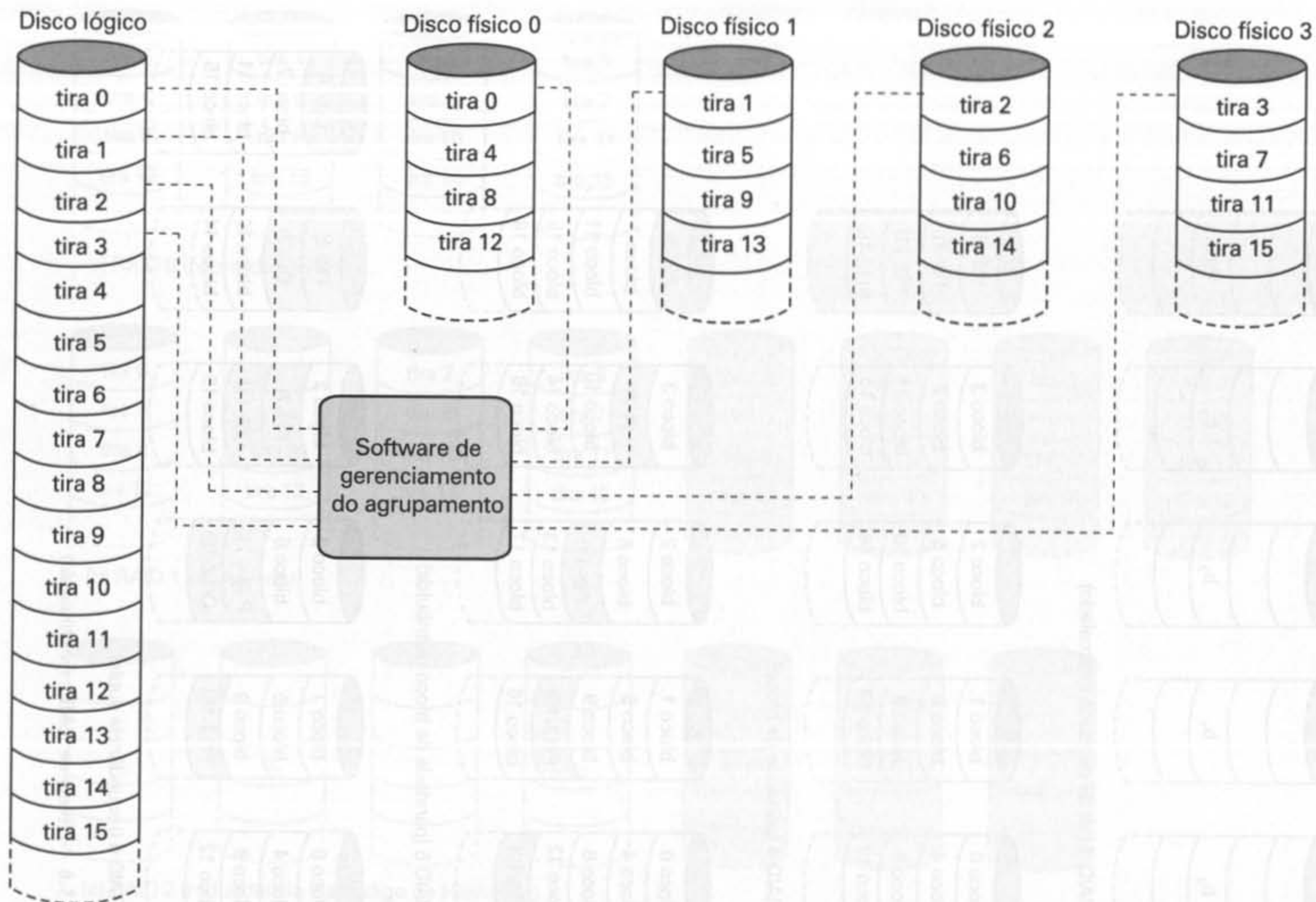


Figura 5.7 Mapeamento de dados para um agrupamento RAID de nível 0.

A segunda condição é que as requisições de E/S feitas pelas aplicações devem utilizar o agrupamento de discos de maneira eficiente. Ela é satisfeita se uma requisição de E/S típica manipula uma quantidade de dados logicamente contíguos bem maior do que a quantidade de dados em uma tira. Nesse caso, a requisição de E/S envolve a transferência paralela de dados contidos em vários discos, com uma taxa de transferência efetiva bem maior do que no caso de um único disco.

RAID 0 para alta taxa de requisição de E/S

Em um ambiente orientado para transações, o usuário tipicamente está mais preocupado com o tempo de resposta do que com a taxa de transferência de dados. Para uma requisição individual de E/S para uma pequena quantidade de dados, o tempo de E/S é dominado pela movimentação dos cabeçotes do disco (tempo de busca) e pelo movimento do disco (latência rotacional).

Em um ambiente orientado para transações, pode haver centenas de requisições de E/S por segundo. Um agrupamento de discos pode fornecer altas taxas de execução de operações de E/S, distribuindo a carga de E/S por meio dos vários discos; esse procedimento também é conhecido como balanceamento de carga. Um balanceamento de carga efetivo só pode ser obtido se houver, tipicamente, diversas requisições de E/S pendentes. Isso significa que devem existir múltiplas aplicações independentes ou uma única aplicação orientada para transações, capaz de submeter múltiplas requisições de E/S assíncronas. O desempenho é também influenciado pelo tamanho da tira. Se o tamanho da tira for relativamente grande, de modo que uma requisição de E/S envolva um único acesso a disco, então as múltiplas requisições de E/S pendentes poderão ser processadas em paralelo, reduzindo o tempo de espera na fila para cada requisição.

RAID de nível 1

O RAID 1 difere dos RAIDs de níveis 2 a 6 pela maneira como a redundância é obtida. Nesses outros níveis, é utilizado algum tipo de cálculo de paridade para introduzir redundância. No RAID 1, a redundância é obtida pela simples duplicação dos dados. Como mostra a Figura 5.6b, os dados são intercalados em tiras, como no RAID 0. Entretanto, nesse caso, cada tira lógica é mapeada em dois discos físicos separados, de modo que cada disco do agrupamento tenha como espelho um outro disco que contém os mesmos dados.

A organização do RAID 1 tem diversos aspectos positivos:

1. Uma requisição de leitura pode ser servida por qualquer dos dois discos que contenha os dados requeridos, preferencialmente por aquele no qual o tempo de busca somado à latência rotacional seja menor.
2. Uma requisição de escrita requer a atualização das duas tiras correspondentes, mas isso pode ser feito em paralelo. Dessa maneira, o desempenho da requisição de escrita é determinado pela operação de escrita mais lenta (por exemplo, aquela que envolve maior tempo de busca e latência rotacional). Não existe qualquer "penalidade de escrita" no RAID 1. Os RAIDs de níveis 2 a 6 envolvem o uso de bits de paridade. Por isso, quando uma tira é atualizada, o software de gerenciamento do agrupamento deve calcular e atualizar os bits de paridade, além de atualizar a tira em questão.
3. A recuperação de uma falha é simples. Quando ocorre uma falha em uma unidade de disco, os dados ainda podem ser obtidos a partir da segunda unidade.

A principal desvantagem do RAID 1 é o custo; ele requer um espaço de disco físico igual a duas vezes o do disco lógico. Por isso, uma configuração RAID 1 geralmente é utilizada apenas em unidades de disco que armazenem software e dados do sistema e outros arquivos altamente críticos. Nesses casos, o RAID 1 oferece uma cópia de segurança de todos os dados em tempo real, de modo que, mesmo ocorrendo uma falha em um disco, todos os dados críticos permaneçam disponíveis.

Em um ambiente orientado para transações, o RAID 1 pode alcançar uma alta taxa de execução de requisições de E/S, caso a maioria das requisições seja de leitura. Nesse caso, o desempenho do RAID 1 pode se aproximar do dobro do desempenho do RAID 0. Entretanto, se uma fração substancial das requisições de E/S for de escrita, pode não haver um ganho significativo de desempenho em relação ao RAID 0. O RAID 1 pode também apresentar um desempenho melhor que o RAID 0 para aplicações com transferência de dados intensiva e com alta porcentagem de leituras. Essa melhoria de desempenho ocorre somente se a aplicação puder dividir cada requisição de leitura, de modo que os dois discos membros possam ser utilizados.

RAID de nível 2

Os RAIDs de níveis 2 e 3 usam a técnica de acesso paralelo. Em um agrupamento com acesso paralelo, todos os discos participam da execução de qualquer requisição de E/S. Tipicamente, os eixos das unidades de disco são sincronizados, de modo que, em qualquer instante, os cabeçotes de todos os discos estejam na mesma posição.

Assim como nos demais esquemas RAID, é usada intercalação de dados em tiras. Nos RAIDs de níveis 2 e 3, as tiras são muito pequenas, freqüentemente do tamanho de um byte ou uma palavra. No RAID 2, um código de correção de erros é calculado para os bits correspondentes de cada disco de dados e os bits do código são armazenados em posições de bit correspondentes nos vários discos de paridade. Tipicamente, é usado um código de Hamming capaz de corrigir um erro em um único bit e de detectar um erro em dois bits.

Embora o RAID 2 exija um número de discos menor que o RAID 1, ele ainda é muito caro. O número de discos redundantes necessários é proporcional ao logaritmo do número de discos de dados. Em uma requisição de leitura, todos os discos são acessados simultaneamente. Os dados requisitados e o código de correção de erros são entregues ao controlador do agrupamento. Se houver um erro em um único bit, o controlador pode detectar e corrigir o erro instantaneamente, sem aumentar o tempo de leitura. Em uma requisição de escrita, todos os discos de dados e os discos de paridade devem ser acessados durante a operação de escrita.

O RAID 2 constitui uma boa escolha apenas em ambientes nos quais podem ocorrer muitos erros de disco. Dada a alta confiabilidade de discos individuais e de controladores de disco, o esquema do RAID 2 é excessivo e por isso não é implementado.

RAID de nível 3

O RAID 3 é organizado de maneira similar ao RAID 2. A diferença é que o RAID 3 requer apenas um disco redundante, independentemente do tamanho do agrupamento de discos. O RAID 3 emprega acesso paralelo, com os dados distribuídos em pequenas tiras. Em vez de um código de correção de erros, apenas um bit de paridade simples é utilizado para cada conjunto de bits localizados na mesma posição em todos os discos de dados.

Redundância

No caso de falha de uma unidade de disco, o disco de paridade é acessado e os dados são reconstruídos a partir dos dados dos dispositivos restantes. Quando a unidade defeituosa for substituída, os dados que faltam podem ser restaurados no disco e a nova unidade pode entrar em operação.

A reconstrução de dados é bastante simples. Considere um agrupamento com cinco unidades de disco, onde os discos X0 a X3 contêm dados e X4 é o disco de paridade. A paridade do *i*-ésimo bit é calculada do seguinte modo:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0\ (i)$$

Suponha que tenha ocorrido uma falha na unidade de disco X1. Acrescentando $X4(i) \oplus X1(i)$ a ambos os lados da equação anterior, obtemos:

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(\mathrm{i}) \oplus X0(i)$$

Portanto, o conteúdo de cada tira do disco X1 pode ser regenerado a partir dos conteúdos das tiras correspondentes nos demais discos do agrupamento. Esse princípio vale para os RAIDs de níveis 3 a 6.

No caso de uma falha de disco, todos os dados ainda permanecem disponíveis no que é conhecido como *modo reduzido*. Nesse modo, os dados que faltam são regenerados no momento de uma requisição de leitura, pelo cálculo de uma operação ou-exclusivo. Quando um valor é escrito sobre um agrupamento RAID 3 em modo reduzido, a coerência dos dados e da paridade deve ser mantida para possibilitar posterior regeneração. Para deixar o modo reduzido e retornar à operação normal, o disco defeituoso deve ser substituído e todo o seu conteúdo deve ser regenerado no novo disco.

Desempenho

Como os dados são distribuídos em tiras bem pequenas, o RAID 3 pode alcançar taxas de transferência de dados bastante altas. Qualquer requisição de E/S envolve a transferência paralela de dados de todos os discos de dados. A melhora de desempenho pode ser especialmente notada no caso de grandes transferências de dados. Por outro lado, apenas uma requisição pode ser executada a cada vez. Dessa maneira, em um ambiente orientado para transações, o desempenho é relativamente baixo.

RAID de nível 4

Os RAIDs de níveis 4 a 6 usam a técnica de acesso independente. Em um agrupamento com acesso independente, cada disco opera independentemente, permitindo que requisições de E/S distintas possam ser satisfeitas em paralelo. Por isso, agrupamentos com acesso independente são mais adequados para aplicações que requerem altas taxas de requisições de E/S, não sendo apropriados para aplicações que necessitam de altas taxas de transferência de dados.

Como nos demais esquemas RAID, é usada a intercalação de dados em tiras. Nos RAIDs de níveis 4 a 6, as tiras são relativamente grandes. No RAID 4, uma tira de paridade é calculada bit a bit sobre as tiras correspondentes em cada disco de dados e os bits de paridade são armazenados na tira correspondente do disco de paridade.

No RAID 4, a escrita de pequenas quantidades de dados envolve certa penalidade. Para cada escrita, o software de gerenciamento do agrupamento tem de atualizar não apenas os dados de usuário, mas também os bits de paridade correspondentes. Considere um agrupamento com cinco unidades de disco, onde os discos X0 a X3 contêm dados e X4 é o disco de paridade. Suponha que seja executada uma operação de escrita que envolve apenas uma tira do disco X1. Inicialmente, para cada bit *i*, temos a seguinte relação:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

Depois da atualização, temos a seguinte relação, onde os bits potencialmente alterados são indicados pelo sinal plica ('):

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

Para calcular a nova paridade, o software de gerenciamento do agrupamento tem de ler a tira antiga de dados de usuário e a tira antiga de paridade. Essas duas tiras são então atualizadas com novos dados e com a paridade recalculada. Dessa maneira, cada operação de escrita de uma tira envolve a realização de duas operações de leitura e duas de escrita.

No caso de uma requisição de escrita de tamanho maior, que envolva tiras de todos os discos, a paridade é facilmente calculada usando apenas os novos bits de dados. Desse modo, o disco de paridade pode ser atualizado em paralelo com os discos de dados, não havendo nenhuma leitura ou escrita extra.

Nos dois casos, toda operação de escrita envolve o disco de paridade, o que pode assim se tornar um gargalo no sistema.

RAID de nível 5

O RAID 5 é organizado de modo semelhante ao RAID 4. A diferença é que o RAID 5 distribui as tiras de paridade por todos os discos. Uma alocação típica consiste em um esquema circular, como mostrado na Figura 5.6f. Para um agrupamento de *n* discos, a tira de paridade das *n* primeiras tiras de dados é armazenada em um disco diferente e esse padrão então se repete.

A distribuição das tiras de paridade em todos os discos evita a possibilidade de formação de gargalos no desempenho do sistema, existentes no RAID 4.

RAID de nível 6

O RAID 6 foi introduzido em um artigo subseqüente pelos pesquisadores de Berkeley (Katz, 1989). No esquema do RAID 6, são usados dois cálculos de paridade diferentes e os resultados são armazenados em blocos separados em discos distintos. Dessa maneira, um agrupamento de RAID 6 no qual os dados de usuário requerem *N* discos é constituído de $N + 2$ discos.

A Figura 5.6g ilustra esse esquema. *P* e *Q* são dois algoritmos distintos de verificação de dados. Um deles consiste em calcular o resultado da operação ou-exclusivo, usada nos RAIDs de níveis 4 e 5. O outro é um algoritmo de verificação independente. Isso torna possível regenerar os dados mesmo no caso de ocorrência de falha em dois discos de dados.

A vantagem do RAID 6 é que apresenta uma disponibilidade de dados extremamente alta, reduzindo-se assim a possibilidade de os dados armazenados serem perdidos. Para que os dados sejam perdidos, é necessário ocorrer uma falha em três discos, dentro do intervalo de tempo médio requerido para reparo*. No entanto, o RAID 6 envolve uma penalidade substancial em operações de escrita, pois cada escrita afeta dois blocos de paridade.

5.3 MEMÓRIA ÓPTICA

Em 1983, foi introduzido um dos mais bem-sucedidos produtos comerciais de todos os tempos: o sistema de áudio digital de disco compacto (CD). O CD é um disco que não pode ser apagado e tem capacidade para armazenar mais de 60 minutos de informação de áudio de cada lado. O enorme sucesso comercial do CD possibilitou o desenvolvimento da tecnologia de armazenamento de discos ópticos de baixo custo, que revolucionou o armazenamento de dados em computadores. Uma variedade de sistemas de disco óptico foi introduzida (Tabela 5.3). A seguir descrevemos cada uma delas de maneira sucinta.

CD-ROM

O CD de áudio e o CD-ROM (*compact disk read-only memory* — disco compacto de memória apenas de leitura) usam tecnologias similares. A principal diferença é que o dispositivo de leitura de um CD-ROM é mais resistente e possui mecanismo de correção de erros para assegurar que os dados sejam transferidos corretamente do disco para o computador. Os dois tipos de disco são fabricados do mesmo modo. O disco é constituído de uma resina do tipo policarbonato e revestido com uma superfície com alto índice de reflexão, normalmente de alumínio. A informação digital registrada (tanto música quanto outros dados) é impressa nessa superfície refletiva como uma série de sulcos microscópicos. A gravação é feita usando, primeiramente, um laser de alta intensidade muito bem focado para criar um disco matriz. Essa matriz é utilizada para fazer um molde para estampar as cópias. A superfície sulcada das cópias é protegida contra pó e arranhões por uma cobertura de laca ou verniz clara.

A informação gravada em um CD ou CD-ROM é lida por um feixe de laser de baixa potência, acondicionado no dispositivo de leitura de disco óptico. O feixe de laser passa através da cobertura protetora transparente, enquanto um motor gira o disco. Ao encontrar um sulco, a intensidade da luz refletida do laser muda. Essa mudança é detectada por um fotossensor e convertida em um sinal digital.

* N.R.T.: O termo técnico da área de confiabilidade de sistemas é tempo médio para o reparo (*mean time to repair* — MTTR), que especifica o tempo médio para que uma unidade falha seja reparada ou substituída. Ou seja, haverá perda de dados no RAID 6 se o tempo entre a primeira e a terceira falha for menor que o intervalo de tempo definido pelo MTTR de disco.

Tabela 5.3 Sistemas de disco óptico

CD Disco compacto. Disco que não pode ser apagado e que armazena informações de áudio digitalizadas. O sistema padrão utiliza discos de 12 cm e pode armazenar mais de 60 minutos ininterruptos de informações de áudio.
CD-ROM Disco compacto de memória de apenas leitura. Disco que não pode ser apagado, usado para armazenar dados de computador. O sistema padrão utiliza discos de 12 cm e pode conter mais de 600 Mbytes.
DVD Disco de vídeo digital (conhecido também como disco versátil digital). É uma tecnologia para representação de informação de vídeo digitalizada e compactada, assim como de grandes volumes de outros dados digitais.
WORM Escrita única — várias leituras (*write-once read-many*). Pode ser gravado mais facilmente do que o CD-ROM, tornando os discos de cópia única comercialmente viáveis. Assim como o CD-ROM, após a gravação o disco pode ser usado apenas para leitura. O tamanho mais comum é de 5,25 polegadas, podendo conter de 200 a 800 Mbytes de dados.
Disco óptico apagável Disco que usa tecnologia óptica, mas pode ser apagado e gravado novamente. Tanto o disco de 3,25 quanto o de 5,25 polegadas estão em uso. A capacidade típica é de 650 Mbytes.
Disco magneto-óptico Disco que usa tecnologia óptica para leitura e técnicas de gravação magnéticas, auxiliadas por focalização óptica. Tanto o disco de 3,25 quanto o de 5,25 polegadas estão em uso. Capacidades acima de 1 Gbyte são comuns.

Como o movimento de um sulco próximo ao centro do disco rotativo, até um determinado ponto fixo (tal como o feixe de laser), é mais lento do que o de um sulco mais externo, essa variação de velocidade deve ser compensada para que a taxa de leitura de todos os sulcos pelo laser seja a mesma. Nos discos magnéticos, essa compensação é feita pelo aumento do espaçamento entre os bits gravados nos segmentos do disco. A informação pode então ser lida a uma taxa constante, girando o disco a uma velocidade fixa, denominada **velocidade angular constante** (*constant angular velocity* — CAV). A Figura 5.8a mostra o esquema de um disco com CAV. O disco é dividido em vários setores circulares de forma trapezoidal e em uma série de trilhas concêntricas. A vantagem do usar a CAV é que blocos individuais de dados podem ser diretamente endereçados por trilha e setor. Para mover o cabeçote de sua posição corrente para um endereço específico, é necessário apenas um pequeno movimento do cabeçote para uma trilha específica e um pequeno tempo de espera até que o setor se encontre sob o cabeçote. A desvantagem de uso da CAV é que a quantidade de dados que pode ser armazenada nas trilhas externas, mais longas, é igual àquela que pode ser armazenada nas trilhas internas, mais curtas.

Como a gravação de menor quantidade de informação nas trilhas externas do disco constitui um desperdício de espaço, o método CAV não é usado em CDs e CD-ROMs. Em vez disso, os dados são distribuídos no disco em segmentos de mesmo tamanho e lidos a uma taxa constante, girando o disco a uma velocidade variável. Os sulcos são lidos pelo laser a uma **velocidade linear constante** (*constant linear velocity* — CLV). O disco gira mais lentamente para acessos próximos à borda externa do disco do que para acessos próximos ao centro. Dessa maneira, a capacidade de uma trilha e o atraso rotacional são maiores para trilhas mais próximas da borda externa do disco.

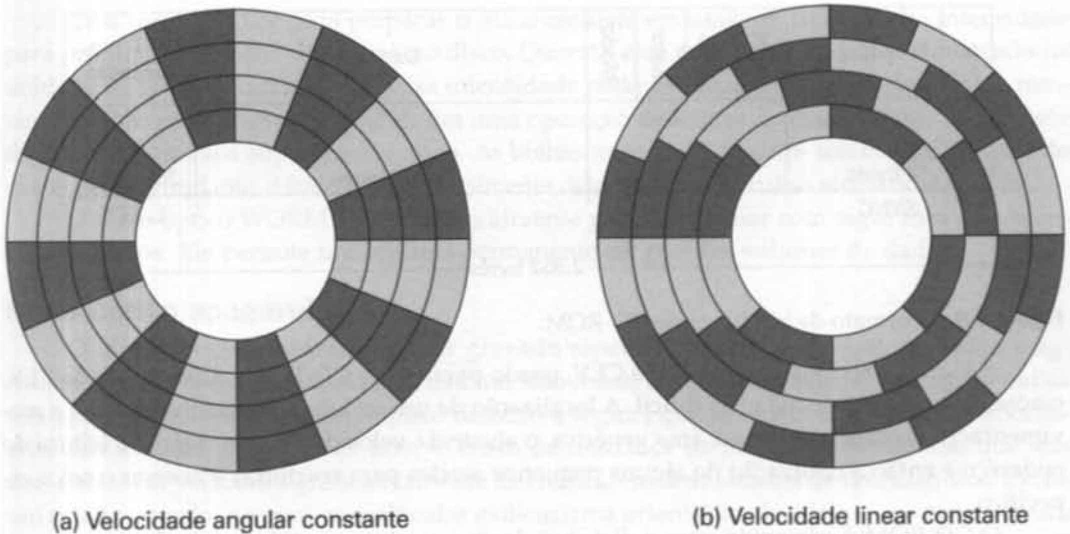


Figura 5.8 Comparação dos métodos de organização de discos.

Os CD-ROMs têm sido produzidos com várias densidades. Em um exemplo típico, o CD-ROM tem espaçamento entre trilhas de 1,6 μm (1,6 x 10^{-6} m). A largura útil na direção radial, que pode ser gravada, é de 32,55 mm, de modo que o número total de trilhas seja igual a 32.550 μm dividido pelo espaçamento entre trilhas, ou seja, 20.344 trilhas. Na verdade, existe uma única trilha espiral, cuja extensão é obtida multiplicando-se o raio médio da circunferência pelo número de voltas da espiral; esse valor é aproximadamente igual a 5,27 km. A velocidade linear constante do CD-ROM é de 1,2 m/s, totalizando 4.391 segundos ou 73,2 minutos de gravação, que é aproximadamente o tempo máximo de execução padrão de um CD de áudio. Como os dados são lidos do disco a 176,4 Kbytes/s, a capacidade de armazenamento do CD-ROM é de 774,57 Mbytes. Isso equivale a mais de 550 disquetes de 3,25 polegadas.

Os dados são organizados no CD-ROM como uma seqüência de blocos. O formato típico de um bloco é mostrado na Figura 5.9. Ele é composto dos seguintes campos:

- **Sync:** o campo sync (sincronismo) identifica o início de um bloco. Ele consiste em um byte com todos os bits de valor 0, seguido de 10 bytes onde todos os bits têm valor 1 e de mais um byte com todos os bits de valor 0.
- **Cabeçalho:** o cabeçalho contém duas partes: o endereço do bloco e um byte indicando um modo. O modo 0 especifica um campo de dados em branco; o modo 1, o uso de código de correção de erros e 2.048 bytes de dados; o modo 2, 2.336 bytes de dados de usuário sem código de correção de erros.
- **Dados:** dados de usuário (2.048 bytes).
- **Auxiliar:** dados de usuário adicionais no modo 2. No modo 1, contém o código de correção de erros de 288 bytes.

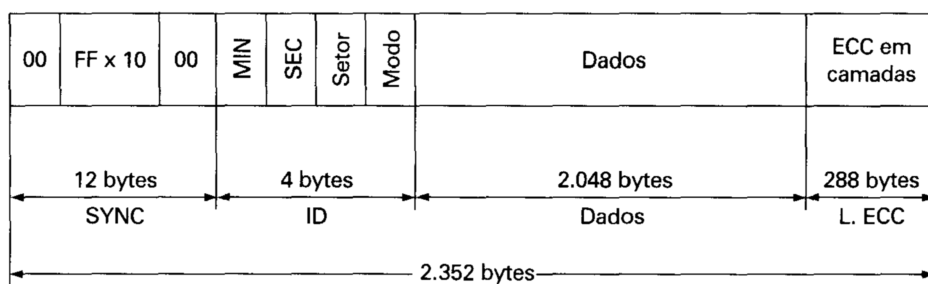


Figura 5.9 Formato de um bloco de CD-ROM.

A Figura 5.8b mostra o esquema CLV, usado para CDs e CD-ROMs. Com o uso de CLV, o acesso aleatório se torna mais difícil. A localização de um endereço específico envolve a movimentação do cabeçote para a área genérica, o ajuste da velocidade de rotação e a leitura do endereço e então a realização de alguns pequenos ajustes para encontrar e acessar o setor específico.

O CD-ROM é adequado para a distribuição de grandes quantidades de dados para um grande número de usuários. Em função do custo inicial do processo de gravação, ele não é adequado para aplicações individualizadas. O CD-ROM possui as seguintes vantagens principais em relação aos discos magnéticos tradicionais:

- A capacidade de armazenamento de dados é muito maior no disco óptico.
- Os dados gravados em um disco óptico podem ser copiados em grande quantidade a um custo baixo, diferentemente do disco magnético. A base de dados em um disco magnético tem de ser reproduzida pela cópia de um disco de cada vez, utilizando duas unidades de disco.
- O disco óptico é removível, permitindo que o próprio disco seja usado como cópia de segurança para arquivos. A maioria dos discos magnéticos não é removível. Os dados gravados em discos magnéticos não-removíveis devem ser primeiramente copiados para uma fita, antes que o disco possa ser utilizado para armazenar outros dados.

As desvantagens de um CD-ROM são:

- Ele só pode ser usado apenas para leitura e não pode ser atualizado.
- O tempo de acesso, da ordem de meio segundo, é muito maior do que o de um disco magnético.

WORM

Para acomodar aplicações nas quais é necessária apenas uma cópia ou um pequeno número de cópias de um conjunto de dados, foi desenvolvido o WORM, um CD que pode ser escrito uma única vez e lido várias vezes. No WORM, o disco é preparado usando um feixe de luz de intensidade moderada, de modo que operações de escrita possam ser realizadas posteriormente. Dessa maneira, com um controlador de disco relativamente mais caro do que um CD-ROM, o cliente pode tanto gravar uma vez quanto ler dados do disco. Para possibilitar um acesso mais rápido, o WORM usa uma velocidade angular constante, com certa perda de capacidade.

Uma técnica típica para preparar o disco consiste em usar um laser de alta intensidade para produzir uma série de bolhas no disco. Quando esse meio pré-formatado é colocado na unidade de WORM, um laser de baixa intensidade pode produzir o calor suficiente para romper as bolhas previamente gravadas. Em uma operação de leitura do disco, o laser da unidade de WORM ilumina a superfície do disco. As bolhas rompidas fornecem um contraste maior do que a área circundante, e isso pode ser facilmente detectado por circuitos eletrônicos simples.

O disco óptico WORM é uma opção atraente para armazenar com segurança documentos e arquivos. Ele permite um registro permanente de grandes volumes de dados.

Disco óptico apagável

O disco óptico apagável pode ser gravado repetidas vezes, como qualquer disco magnético. Embora diversas tentativas já tenham sido feitas, a única abordagem puramente óptica (em oposição ao disco magneto-óptico descrito a seguir) que se mostrou atraente é a abordagem denominada mudança de fase. O disco de mudança de fase usa um material que tem dois índices de reflexão significativamente diferentes, em dois estados de fase distintos. Existe um estado amorfo, no qual as moléculas exibem uma orientação aleatória e que reflete pouca luz; e um estado cristalino, que possui uma superfície suave com boa reflexão de luz. Um feixe de luz laser pode alterar o material de uma fase para outra. A principal desvantagem dos discos ópticos de mudança de fase é que seu material eventualmente perde de maneira permanente essa propriedade. Os materiais atualmente existentes podem ser apagados 500 mil a 1 milhão de vezes.

O disco óptico apagável tem sobre o CD-ROM e o WORM a vantagem de poder ser regravado e, portanto, usado como memória secundária. Como tal, ele compete com o disco magnético. As principais vantagens do disco óptico apagável em relação ao disco magnético são:

- **Alta capacidade:** um disco óptico de 5,25 polegadas pode conter cerca de 650 Mbytes de dados. A capacidade de alguns discos Winchester é de menos que a metade desse valor.
- **Portabilidade:** o disco óptico pode ser removido da unidade controladora.
- **Confiabilidade:** a tolerância de parâmetros de engenharia de discos ópticos é muito menos severa do que no caso de discos magnéticos de alta capacidade. Portanto, eles apresentam maior confiabilidade e um tempo de vida mais longo.

Assim como com o WORM, o disco óptico apagável usa velocidade angular constante.

Disco de vídeo digital

Com o disco de vídeo digital de grande capacidade (DVD), a indústria eletrônica finalmente encontrou um substituto aceitável para as fitas de vídeo VHS analógicas. O DVD substituirá tanto a fita de vídeo usada em videocassetes (VCRs) quanto o CD-ROM nos computadores pessoais e servidores. Ele traz o vídeo até a era digital. Possibilita a exibição de filmes com uma qualidade de imagem impressionante e o acesso pode ser feito aleatoriamente, como nos CDs de áudio, que também podem ser utilizados nos dispositivos de DVD. Grandes volumes de dados podem ser gravados em um disco: atualmente até sete vezes a quantidade que pode ser gravada em um CD-ROM. Em virtude de sua enorme capacidade de armazenamento e da imagem extremamente límpida, os jogos desenvolvidos para microcomputadores se tornam mais realistas e uma quantidade maior de vídeos pode ser incorporada aos softwares educa-

cionais. A inclusão desse tipo de material nos sites Web deverá ocasionar um novo pico de tráfego na Internet, assim como em redes intranet corporativas.

As principais características que distinguem o DVD do CD-ROM são:

- Um DVD padrão contém 4,7 Gbytes por camada, e um DVD de um único lado e camada dupla contém 8,5 Gbytes.
- O DVD usa uma forma de compressão de vídeo para imagens de alta qualidade que ocupam a tela inteira, conhecida como MPEG.
- Um DVD de uma única camada pode conter um filme de duas horas e meia, enquanto um DVD de camada dupla pode conter um filme de mais de quatro horas de duração.

Disco magneto-óptico

Uma unidade de disco magneto-óptico (MO) usa um laser óptico para aumentar a capacidade do sistema de disco magnético convencional. A tecnologia de gravação é fundamentalmente magnética. Entretanto, um laser óptico é usado para focalizar o cabeçote de gravação magnética, possibilitando obter maior capacidade. Nesse esquema, o disco é coberto com um material cuja polaridade pode ser alterada apenas a altas temperaturas. A informação é gravada no disco utilizando o laser para aquecer um ponto mínimo da superfície e então aplicar o campo magnético. Quando esse ponto esfria, ele adota a polaridade de campo norte-sul. Como esse processo de polarização não causa mudanças físicas no disco, ele pode ser repetido várias vezes.

A operação de leitura é puramente óptica. A direção do magnetismo pode ser detectada por uma luz de laser polarizada (com potência mais baixa do que a necessária para a operação de escrita). A luz polarizada que é refletida de um ponto particular muda seu grau de rotação dependendo da orientação do campo magnético nesse ponto.

A principal vantagem do disco MO sobre o CD puramente óptico é sua longevidade. Repetidas regravações no disco óptico resultam em degradação gradual do meio. O disco MO não apresenta essa degradação e, portanto, pode ser usado para um número muito maior de operações de regravação. Outra vantagem da tecnologia MO é seu custo por megabyte, consideravelmente mais baixo do que no caso do disco magnético.

5.4 FITA MAGNÉTICA

Os sistemas de fita usam as mesmas técnicas de leitura e gravação dos sistemas de discos. O meio consiste em uma fita Mylar flexível, coberta com óxido magnético. A fita e a unidade de fita são análogas ao de um sistema de gravação de fita comum.

A fita é organizada como um pequeno número de trilhas paralelas. Os sistemas de fita antigos usavam, tipicamente, nove trilhas. Isso tornava possível armazenar um byte de cada vez, com um bit de paridade adicional na nona trilha. Sistemas de fita mais modernos usam 18 ou 36 trilhas, correspondendo a uma palavra ou a uma palavra dupla. Assim como no disco, os dados são lidos e escritos na fita em blocos contíguos, denominados *registros físicos*. Os blocos da fita são separados por espaços denominados *espaços entre registros* (*interrecord gaps* ou IRGs). A Figura 5.10 mostra a estrutura de uma fita de nove trilhas. Assim como o disco, a fita é formatada de modo que torne mais fácil a localização de registros físicos.

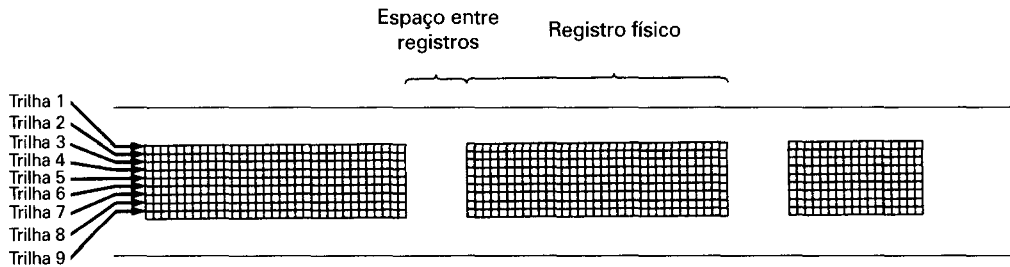


Figura 5.10 Formato de uma fita magnética de nove trilhas.

Uma unidade de fita é um dispositivo de *acesso seqüencial*. Se a cabeça da fita estiver posicionada no registro 1, para ler o registro N será necessário ler os registros físicos de 1 a $N - 1$, um de cada vez. Se a cabeça estiver posicionada além do registro desejado, será preciso rebobinar a fita até certo ponto e iniciar a leitura a partir desse ponto. Diferentemente do disco, a fita se move apenas durante uma operação de leitura ou escrita.

A unidade de disco, ao contrário da fita, é um dispositivo de *acesso direto*. Ela não precisa ler todos os setores do disco seqüencialmente para chegar ao setor desejado. Deve apenas esperar os setores desejados dentro de uma trilha e pode fazer acessos sucessivos a qualquer trilha.

A fita magnética foi o primeiro tipo de memória secundária. Ela é ainda largamente utilizada como o elemento de menor custo e menor velocidade da hierarquia de memória.

5.5 LEITURA E SITES WEB RECOMENDADOS

Uma boa abordagem sobre a tecnologia básica de gravação de sistemas de fita e de disco pode ser encontrada em Mee (1996a). Mee (1996b) enfoca técnicas de armazenamento de dados de sistemas de disco e de fita.

Chen (1994) apresenta uma excelente visão geral sobre a tecnologia RAID, escrita por seus próprios inventores. Uma discussão mais detalhada foi publicada pela RAID Advisory Board, uma associação de fornecedores e consumidores de produtos relacionados a RAID (Massiglia, 1997). E um bom artigo recente é o de Friedman (1996).

Uma excelente visão geral da área de armazenamento óptico é apresentada por Marchant (1990). E uma boa descrição da tecnologia de gravação e leitura em dispositivos ópticos pode ser encontrada em Mansuripur (1997).

Finalmente, Rosch (1997) apresenta uma descrição ampla sobre todos os tipos de sistemas de memória externa, com pequena quantidade de detalhes técnicos sobre cada uma.



Site Web recomendado:

- **RAID Advisory Board:** grupo da Indústria RAID.

5.6 EXERCÍCIOS

5.1 Defina os seguintes parâmetros para um sistema de disco:

t_S = tempo de busca; tempo médio para posicionar o cabeçote sobre a trilha
r = velocidade de rotação do disco, em revoluções por segundo
n = número de bits por setor
N = capacidade de uma trilha em bits
t_A = tempo de acesso a um setor

Obtenha uma fórmula para t_A como função dos demais parâmetros.

5.2 Considere uma configuração RAID com dez unidades de disco. Preencha a tabela a seguir, que compara os vários níveis de RAID:

Nível de RAID	Densidade de armazenamento	Desempenho de largura de banda	Desempenho de transação
0	1		1
1			
2			
3		1	
4			
5			

Cada parâmetro é normalizado em relação ao nível de RAID que oferece o melhor desempenho; portanto, os demais números a serem preenchidos na tabela devem ter valor entre 0 e 1. A densidade de armazenamento é a fração da área de armazenamento do disco disponível para os dados de usuário. O desempenho de largura de banda reflete a velocidade de transferência de dados de um agrupamento. O desempenho de transação indica o número de operações de E/S que podem ser executadas pelo agrupamento por segundo.

5.3 A intercalação de dados em tiras no disco pode melhorar a taxa de transferência de dados, quando o tamanho da tira é menor do que o da requisição de E/S. O RAID 0 apresenta um desempenho melhor do que o de um único disco grande, uma vez que várias requisições de E/S podem ser realizadas em paralelo. Nesse caso, a intercalação de dados no disco seria ainda necessária? Em outras palavras, a taxa de requisição de E/S em um agrupamento de discos que usa intercalação de dados em filetes é maior do que em um agrupamento de discos que não utiliza essa técnica?

5.4 Qual é a taxa de transferência de dados de uma unidade de fita magnética com nove trilhas, cuja velocidade é de 120 polegadas por segundo e cuja densidade de fita é de 1.600 bits lineares por polegada?

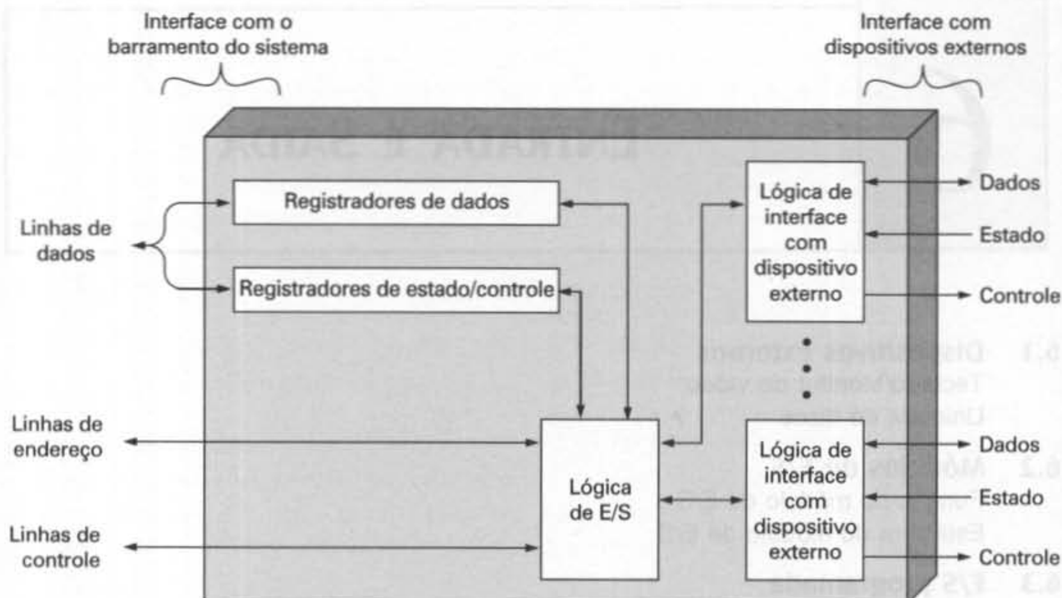
5.5 Considere um carretel de fita de 2.400 pés; um espaço entre registros de 0,6 polegada, no qual a fita pára, entre leituras. Suponha que a taxa de aumento ou diminuição de velocidade da fita nas separações entre registros seja linear e que as demais características da fita sejam as mesmas do Exercício 5.4. Os dados estão organizados na fita em registros físicos, cada qual com um número fixo de unidades de dados de usuário, denominadas registros lógicos.

- **a.** Quanto tempo é necessário para ler toda a fita, se os registros lógicos têm 120 bytes, e cada 10 registros lógicos são agrupados em um registro físico?
- **b.** Quanto tempo é necessário para ler toda a fita, se os registros lógicos têm 120 bytes e cada 30 registros lógicos são agrupados em um registro físico?
- **c.** Quantos registros lógicos a fita contém em cada um dos casos anteriores?
- **d.** Qual é a taxa de transferência efetiva em cada um dos casos anteriores?
- **e.** Qual é a capacidade da fita?

5.6 Calcule o espaço em disco (em setores, trilhas e superfícies) necessário para armazenar os registros lógicos do Exercício 5.5b, considerando um disco com setores de tamanho fixo de 512 bytes/setor, 96 setores/trilha, 110 trilhas por superfície e 8 superfícies úteis. Ignore registros de cabeçalho de arquivos e índices de trilhas e suponha que um registro não possa se estender por mais de um setor.

capítulo

6 ENTRADA E SAÍDA



- A arquitetura de E/S de um computador constitui sua interface com o mundo exterior. Ela é projetada para permitir um controle sistemático da interação com o mundo exterior e fornecer ao sistema operacional as informações de que ele necessita para gerenciar a atividade de E/S de maneira efetiva.
- Existem três técnicas principais de E/S: na **E/S programada**, a E/S é efetuada sob controle direto e contínuo do programa que requisitou a operação de E/S; na **E/S dirigida por interrupção**, o programa envia um comando de E/S e então continua a execução de instruções até que ocorra uma interrupção gerada pelo hardware de E/S, que sinaliza o término da operação de E/S requerida; na técnica de **acesso direto à memória (*direct memory access* — DMA)**, a E/S é controlada por um processador especializado de E/S, que se encarrega de transferir os blocos de dados.
- Dois exemplos importantes de interfaces externas de E/S são as interfaces SCSI (*small computer system interface* — interface de sistemas de computação pequenos) e *FireWire*. A **SCSI** é uma interface paralela para dispositivos externos e a ***FireWire*** é uma interface serial de alta velocidade.

Além do processador e da memória, um terceiro elemento fundamental de um sistema de computação é o conjunto de módulos de E/S. Cada módulo se conecta com o barramento do sistema ou com o comutador central e controla um ou mais dispositivos periféricos. Um módulo de E/S não é simplesmente um conjunto de conectores mecânicos que ligam um dispositivo ao barramento do sistema. Ele contém certa 'inteligência', isto é, uma lógica dedicada a desempenhar a função de comunicação entre o periférico e o barramento.

Você pode se perguntar por que os periféricos não são diretamente conectados ao barramento do sistema. Isso não ocorre pelos seguintes motivos:

- Existe uma grande variedade de periféricos, com diferentes mecanismos de operação. Seria impraticável incorporar ao processador a lógica necessária para controlar vários dispositivos diferentes.
- Como a taxa de transferência de dados dos periféricos é, freqüentemente, muito menor do que a taxa de transferência de dados da memória ou do processador, torna-se impraticável usar barramentos do sistema de alta velocidade para a comunicação direta com um periférico.
- Os periféricos usam freqüentemente formatos de dados e tamanhos de palavras diferentes dos usados no computador ao qual estão conectados.

Por essas razões, é requerido um módulo de E/S, que deve desempenhar duas funções principais (Figura 6.1):

- Fornecer uma interface com o processador e a memória, através do barramento do sistema ou do comutador central.
- Permitir a interface com um ou mais dispositivos periféricos, através de conexões de dados adequadas.

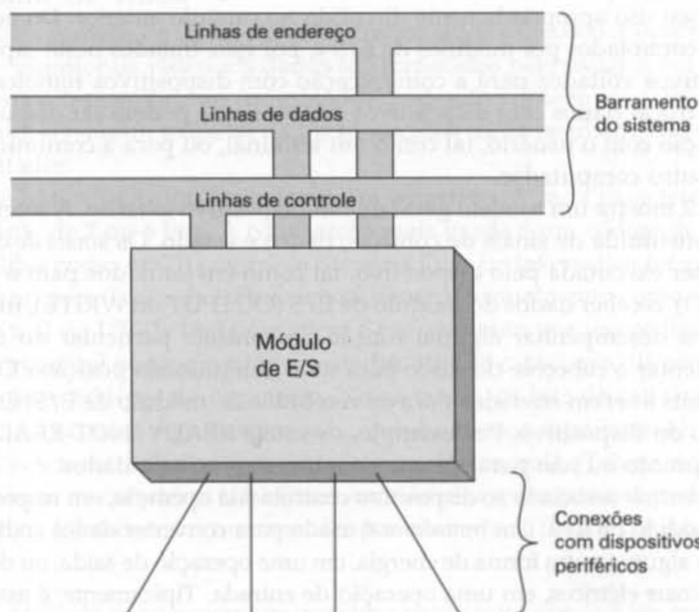


Figura 6.1 Modelo geral de um módulo de E/S.

Este capítulo começa com uma breve discussão sobre dispositivos externos e uma visão geral da estrutura e da função de um módulo de E/S. Em seguida, são abordados os diversos métodos com os quais a função de E/S pode ser realizada em cooperação com o processador e a memória: a interface interna de E/S. Finalmente, é examinada a interface entre o módulo de E/S e o mundo exterior: a interface externa de E/S.

6.1 DISPOSITIVOS EXTERNOS

As operações de E/S são efetuadas por meio de grande variedade de dispositivos externos, que oferecem um meio para a troca de dados entre o ambiente externo e o computador. Um dispositivo externo é conectado ao computador através de uma conexão de um módulo de E/S (Figura 6.1). Essa conexão é usada para a transferência de dados, informações de controle e informações de estado entre o módulo de E/S e o dispositivo externo. Um dispositivo externo conectado a um módulo de E/S é freqüentemente denominado *dispositivo periférico* ou, simplesmente, *periférico*.

Os dispositivos externos podem ser classificados em três categorias:

- Dispositivos voltados para a comunicação com o usuário.
- Dispositivos voltados para a comunicação com a máquina.
- Dispositivos voltados para a comunicação com dispositivos remotos.

Alguns exemplos de dispositivos voltados para a comunicação com o usuário são os terminais de vídeo (VDTs) e impressoras. Dispositivos voltados para a comunicação com a máquina são, por exemplo, os discos magnéticos e os sistemas de fitas, e os sensores e os controladores usados em aplicações de robótica. Note que os sistemas de disco e de fita são vistos neste capítulo como dispositivos de E/S, ao passo que no Capítulo 5 eles são vistos como dispositivos de memória. Do ponto de vista funcional, esses dispositivos são parte da hierarquia de memória, sendo seu uso apropriadamente discutido no capítulo anterior. Do ponto de vista estrutural, eles são controlados por módulos de E/S e, por isso, tratados neste capítulo.

Os dispositivos voltados para a comunicação com dispositivos remotos possibilitam a um computador trocar dados com dispositivos remotos, que podem ser dispositivos voltados para a comunicação com o usuário, tal como um terminal, ou para a comunicação interna ou mesmo ser um outro computador.

A Figura 6.2 mostra um modelo geral de um dispositivo externo. A interface com o módulo de E/S é constituída de sinais de controle, dados e estado. Os *sinais de controle* determinam a função a ser executada pelo dispositivo, tal como enviar dados para o módulo de E/S (INPUT ou READ), receber dados do módulo de E/S (OUTPUT ou WRITE), informar o estado do dispositivo ou desempenhar alguma função de controle particular do dispositivo (por exemplo, movimentar o cabeçote do disco para uma determinada posição). Os *dados* formam um conjunto de bits a serem enviados para ou recebidos do módulo de E/S. Os *sinais de estado* indicam o estado do dispositivo. Por exemplo, os sinais READY/NOT-READY indicam se o dispositivo está pronto ou não para efetuar uma transferência de dados.

A *lógica de controle* associada ao dispositivo controla sua operação, em resposta a um comando recebido do módulo de E/S. Um *transdutor* é usado para converter dados codificados como sinais elétricos para alguma outra forma de energia, em uma operação de saída, ou dessa outra forma de energia para sinais elétricos, em uma operação de entrada. Tipicamente, é associada ao transdutor uma área de armazenamento temporário para os dados a serem transferidos entre o módulo de E/S e o ambiente externo; essa área normalmente tem um tamanho de 8 ou 16 bits.

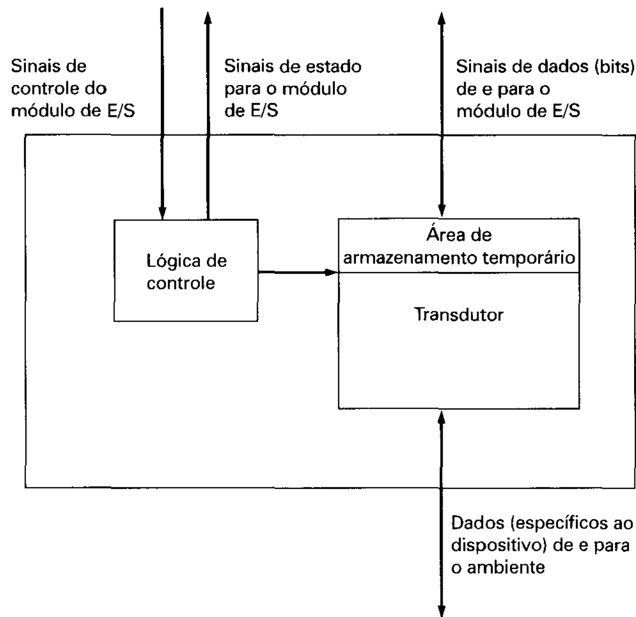


Figura 6.2 Dispositivo externo.

A interface entre o módulo de E/S e o dispositivo externo é tratada na Seção 6.7. Embora a descrição da interface entre os dispositivos externos e o ambiente não faça parte do escopo deste livro, alguns exemplos são descritos brevemente a seguir.

Teclado/monitor de vídeo

A forma mais comum de interação entre computador e usuário ocorre por meio da combinação teclado/monitor de vídeo. Os dados são fornecidos pelo usuário por meio do teclado. Essa entrada é então transmitida ao computador, podendo também ser exibida no monitor de vídeo. Além disso, o monitor exibe os dados fornecidos como resultado de operações realizadas pelo computador.

A unidade básica de troca de dados é um caractere. A cada caractere é associado um código tipicamente de 7 ou 8 bits. A codificação mais usada é um código de 7 bits conhecido nos Estados Unidos como ASCII (*American Standard Code for Information Interchange* — Código Padrão Americano para Troca de Informações) e internacionalmente como Alfabeto de Referência Internacional da ITU-T. Cada caractere é representado por um código binário distinto de 7 bits; dessa maneira, podem ser representados até 128 caracteres diferentes. A Tabela 6.1 relaciona todos esses códigos de caracteres. Nessa tabela, os bits de cada caractere são rotulados de b_7, que é o bit mais significativo, a b_1, que é o menos significativo[1]. Os caracteres são de dois tipos: caracteres imprimíveis e caracteres de controle (Tabela 6.2). Os caracteres imprimíveis incluem os caracteres alfabéticos, numéricos e especiais que podem ser impressos ou exibidos na tela. O código do caractere 'K', por exemplo, é 1001011. Alguns dos caracteres

1. Os caracteres ASCII são quase sempre armazenados e transmitidos usando 8 bits por caractere. O oitavo bit é um bit de paridade, utilizado para detecção de erro. O bit de paridade é o bit mais significativo, rotulado como b_8.

controle servem para controlar a impressão ou a exibição de caracteres; um exemplo é o caractere de retorno de carro. Outros caracteres de controle dizem respeito a procedimentos de comunicação.

Tabela 6.1 O Código ASCII

posição do bit

							0	0	0	0	1	1	1	1
							0	0	1	1	0	0	1	1
							0	1	0	1	0	1	0	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1								
			0	0	0	0	NUL	DLE	SP	0	@	P	`	p
			0	0	0	1	SOH	DC1	!	1	A	Q	a	q
			0	0	1	0	STX	DC2	″	2	B	R	b	r
			0	0	1	1	ETX	DC3	#	3	C	S	c	s
			0	1	0	0	EOT	DC4	$	4	D	T	d	t
			0	1	0	1	ENQ	NAK	%	5	E	U	e	u
			0	1	1	0	ACK	SYN	&	6	F	V	f	v
			0	1	1	1	BEL	ETB	'	7	G	W	g	w
			1	0	0	0	BS	CAN	(	8	H	X	h	x
			1	0	0	1	HT	EM	)	9	I	Y	i	y
			1	0	1	0	LF	SUB	*	:	J	Z	j	z
			1	0	1	1	VT	ESC	+	;	K	[	k	{
			1	1	0	0	FF	FS	,	<	L	\	l	|
			1	1	0	1	CR	GS	-	=	M	]	m	}
			1	1	1	0	SO	RS	.	>	N	^	n	~
			1	1	1	1	SI	US	/	?	O	-	o	DEL

Essa tabela corresponde à versão do Alfabeto de Referência Internacional da ITU-T (T.50), adotada nos Estados Unidos. O significado dos caracteres de controle é explicado na Tabela 6.2.

Quando uma tecla é pressionada pelo usuário, gera-se um sinal eletrônico, que é interpretado pelo transdutor do teclado e traduzido de acordo com o padrão de bits correspondente do código ASCII. Esse padrão de bits é então transmitido para o módulo de E/S do computador. Um texto pode ser armazenado no computador, codificado nesse mesmo código ASCII. Em uma operação de saída de dados, os caracteres em código ASCII são transmitidos para um dispositivo externo por meio do módulo de E/S. O transdutor desse dispositivo interpreta os códigos e envia os sinais eletrônicos correspondentes para o dispositivo de saída, que então exibe o caractere indicado ou desempenha a função de controle requerida.

Unidade de disco

A unidade de disco contém circuitos eletrônicos para a troca de dados e de sinais de controle e de estado com o módulo de E/S, além de circuitos para controlar o mecanismo de leitura/escrita do disco. Em um disco de cabeçote fixo, o transdutor converte padrões magnéticos da superfície do disco para bits da área de armazenamento temporário do dispositivo e vice-versa (Figura 6.2). Um disco de cabeçote móvel deve também ser capaz de mover o braço do disco na direção radial, sobre a superfície do disco.

Tabela 6.2 Caracteres de controle do código ASCII

Controle de formatação	
BS (Retrocesso): indica o movimento do mecanismo de impressão ou do cursor do monitor de vídeo de uma posição para trás. **HT** (Tabulação horizontal): indica o avanço do mecanismo de impressão ou do cursor do monitor de vídeo para a próxima posição de "tabulação" predefinida ou para uma posição final. **LF** (Próxima linha): indica o movimento do mecanismo de impressão ou do cursor do monitor de vídeo para o início da próxima linha.	**VT** (Tabulação vertical): indica o movimento do mecanismo de impressão ou do cursor do monitor de vídeo de um certo número predefinido de linhas. **FF** (Alimentação de formulário): indica o movimento do mecanismo de impressão ou do cursor do monitor de vídeo para a posição inicial da próxima página, formulário ou tela. **CR** (Retorno de carro): indica o movimento do mecanismo de impressão ou do cursor do monitor de vídeo para a posição inicial da linha corrente.
Controle de transmissão	
SOH (Início de cabeçalho): usado para indicar o início de um cabeçalho, que pode conter um endereço ou uma informação de roteamento. **STX** (Início de texto): usado para assinalar o início de um texto, indicando também o fim do seu cabeçalho. **ETX** (Fim de texto): usado para indicar o término do texto iniciado com STX. **EOT** (Fim de transmissão): indica o fim de uma transmissão, que pode incluir um ou mais 'textos' com seus respectivos cabeçalhos. **ENQ** (Consulta): requisita uma resposta de uma estação remota. Pode ser usado como uma requisição do tipo 'QUEM É VOCÊ', que solicita a identificação de uma estação.	**ACK** (Confirmação de recebimento): caractere transmitido por um dispositivo como confirmação de recebimento de uma transmissão efetuada por um dispositivo remetente. Também usado como resposta afirmativa para mensagens de interrogação. **NAK** (Não-confirmação de recebimento): caractere transmitido por um dispositivo receptor como um aviso de não-confirmação de recebimento de uma transmissão efetuada por um dispositivo remetente. Também usado como resposta negativa para mensagens de interrogação de dispositivos (*polling*). **SYN** (Síncrono/Ocioso): usado para obter a sincronização em sistemas de transmissão síncrona. Quando nenhum dado está sendo enviado, um sistema de transmissão síncrona pode enviar caracteres SYN continuamente. **ETB** (Fim de transmissão de bloco): indica o fim da transferência de um bloco de dados. É usado para agrupar dados em blocos, quando a estrutura de blocos não é necessariamente relacionada ao formato de processamento.

(*continua*)

Tabela 6.2 Caracteres de controle do código ASCII (*Continuação*)

Separadores de Informação	
FS (Separador de arquivo) **GS** (Separador de grupo) **RS** (Separador de registro) **US** (Separador de unidade)	Separadores de informação, que devem ser usados em caráter excepcional e cuja hierarquia é no sentido de FS (o mais inclusivo) para US (o menos inclusivo).
Outros caracteres	
NUL (Nulo): caractere que indica ausência. Usado para preenchimento de tempo de transmissão ou de espaço em fita, quando não há dados para serem transmitidos. **BEL** (Sinál sonoro): usado para chamar a atenção do usuário para algum evento. Pode ser usado para controle de alarmes ou outros dispositivos sonoros. **SO** (Sair do conjunto de caracteres padrão): mostra que as combinações de códigos subseqüentes devem ser interpretadas fora do conjunto de caracteres padrão, até que seja enviado um caractere SI. **SI** (Retornar ao conjunto de caracteres padrão): indica que as combinações de códigos subseqüentes devem ser interpretadas de acordo com o conjunto de caracteres padrão. **DEL** (Apagar): usado para apagar caracteres não desejados. **SP** (Espaço): caractere não-imprimível, usado para separar palavras ou para mover o mecanismo de impressão ou o cursor do monitor de vídeo de uma posição para a frente.	**DLE** (Escape de conexão de dados): caractere usado para alterar o significado de um ou mais caracteres subseqüentes. Pode possibilitar controles suplementares ou permitir o envio de caracteres de dados com qualquer combinação de bits. **DC1, DC2, DC3, DC4** (Controles de dispositivos): caracteres para controle de dispositivos auxiliares ou de características especiais de terminais. **CAN** (Cancelamento): indica que os dados que o precedem, em uma mensagem ou um bloco, devem ser ignorados (usualmente porque foi detectado algum erro). **EM** (Fim de um meio): indica o fim físico de uma fita ou algum outro meio ou o fim de uma parte requerida ou usada de um dado meio. **SUB** (Substituição): substitui um caractere inválido ou em que foi detectado um erro. **ESC** (Escape): caractere usado para fornecer extensão do conjunto de códigos de caracteres; especifica que um número específico de caracteres subseqüentes tem um significado diferente do usado sem o caractere ESC.

6.2 MÓDULOS DE E/S

Função do módulo de E/S

As funções mais importantes de um módulo de E/S podem ser divididas nas seguintes categorias:

- Controle e temporização
- Comunicação com o processador
- Comunicação com dispositivos
- Área de armazenamento temporário de dados
- Detecção de erros

O processador pode comunicar-se a qualquer momento com um ou mais dispositivos externos, dependendo das necessidades de E/S do programa. Os recursos internos do sistema, tais como a memória principal e o barramento, são compartilhados para a realização de

diversas atividades, incluindo a E/S de dados. Por isso, um módulo de E/S inclui funções de **controle e temporização**, para controlar o fluxo de dados entre os recursos internos e os dispositivos externos. Por exemplo, o controle de transferência de dados de um dispositivo externo para o processador pode envolver a seguinte seqüência de etapas:

1. O processador interroga o módulo de E/S para verificar o estado do dispositivo a ele conectado.
2. O módulo de E/S retorna o estado do dispositivo.
3. Se o dispositivo estiver em operação e pronto para transmitir, o processador requisitará a transferência de dados, enviando um comando para o módulo de E/S.
4. O módulo de E/S obtém uma unidade de dados (por exemplo, 8 ou 16 bits) do dispositivo externo.
5. Os dados são transferidos do módulo de E/S para o processador.

Se um barramento é usado pelo sistema, cada uma das interações entre o processador e o módulo de E/S envolve uma ou mais arbitrações do barramento.

O cenário simplificado descrito anteriormente mostra também que o módulo de E/S deve ser capaz de comunicar-se tanto com o processador quanto com o dispositivo externo. A **comunicação com o processador** envolve os seguintes tópicos:

- **Decodificação de comando:** o módulo de E/S recebe comandos do processador, enviados tipicamente como sinais, através do barramento de controle. Por exemplo, um módulo de E/S ao qual é conectada uma unidade de disco pode aceitar os seguintes comandos: leitura de setor, escrita de setor, busca de uma determinada trilha e pesquisa por um registro com um determinado identificador. No caso dos dois últimos comandos, um parâmetro é enviado através do barramento de dados.
- **Dados:** os dados são transferidos entre o processador e o módulo de E/S através do barramento de dados.
- **Informação de estado:** como os periféricos são, em geral, muito lentos, é importante conhecer o estado do módulo de E/S. Por exemplo, em uma operação de leitura, o módulo de E/S pode muitas vezes não estar pronto para enviar os dados requeridos para o processador porque ainda está processando o comando de E/S anterior. Essa informação é enviada como um sinal de estado. Alguns sinais de estado comuns são BUSY (ocupado) e READY (pronto). Pode também haver sinais que especificam condições de erro.
- **Reconhecimento de endereço:** assim como cada palavra da memória, cada dispositivo de E/S tem um endereço. Dessa maneira, o módulo de E/S deve reconhecer um endereço distinto para cada periférico que ele controla.

Por outro lado, um módulo de E/S deve ser também capaz de realizar **comunicação com os dispositivos**. Essa comunicação envolve comandos, informação de estado e dados (Figura 6.2).

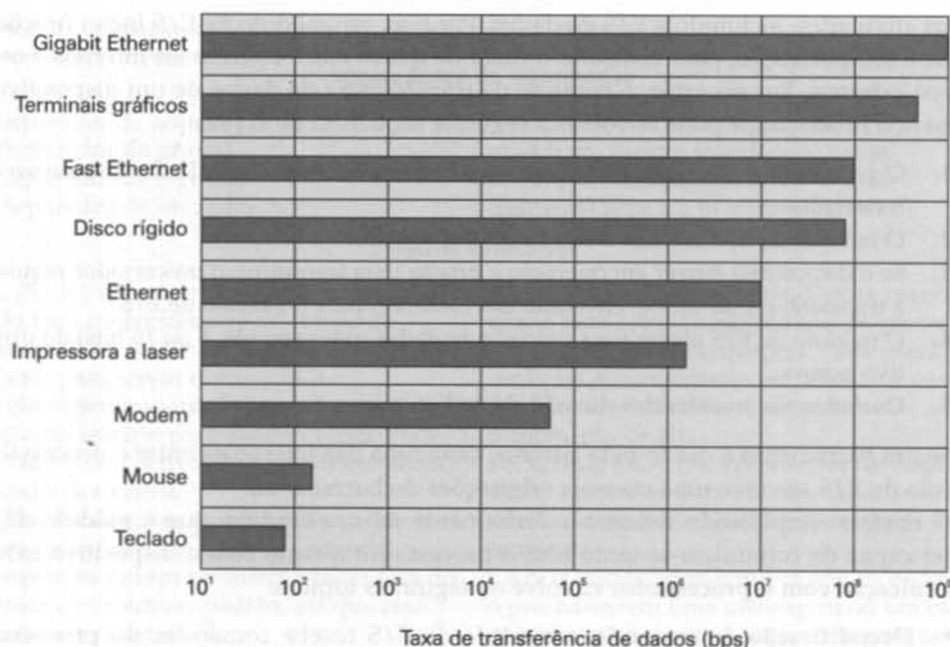


Figura 6.3 Taxas de transferência de dados típicas de dispositivos de E/S.

Uma tarefa essencial de um módulo de E/S é o **armazenamento temporário de dados**. A Figura 6.3 mostra por que essa função é necessária. Enquanto a taxa de transferência de dados entre a memória principal e o processador é bastante alta, as taxas da maioria dos dispositivos periféricos são ordens de grandeza menores e compreendem uma ampla faixa de valores. A transferência de dados da memória principal para o módulo de E/S é feita rapidamente. Esses dados são temporariamente armazenados no módulo de E/S e, então, enviados para o dispositivo periférico em uma taxa adequada. Na transmissão em sentido oposto, os dados são também armazenados temporariamente no módulo de E/S, para não reter a memória em uma transferência de dados a baixa velocidade. Dessa maneira, o módulo de E/S deve ser capaz de realizar operações tanto à velocidade da memória quanto à do dispositivo externo.

Finalmente, um módulo de E/S freqüentemente é responsável pela **detecção de erros** e pelo envio de informações de erro para o processador. Possíveis erros incluem mau funcionamento mecânico ou elétrico sinalizado pelo dispositivo (por exemplo, uma falha de alimentação de papel na impressora ou uma trilha de disco defeituosa), assim como alterações no padrão de bits transmitido por um dispositivo para o módulo de E/S. Para detectar erros de transmissão, é usado algum tipo de código de detecção de erros. Um exemplo comum é o uso de um bit de paridade em cada caractere de dados. Por exemplo, o código de um caractere ASCII ocupa 7 dos 8 bits de um byte e o valor atribuído ao oitavo bit é determinado de modo que o número total de 1s no byte seja par (paridade par) ou ímpar (paridade ímpar). Quando um byte é recebido, o módulo de E/S verifica a paridade para determinar se ocorreu um erro.

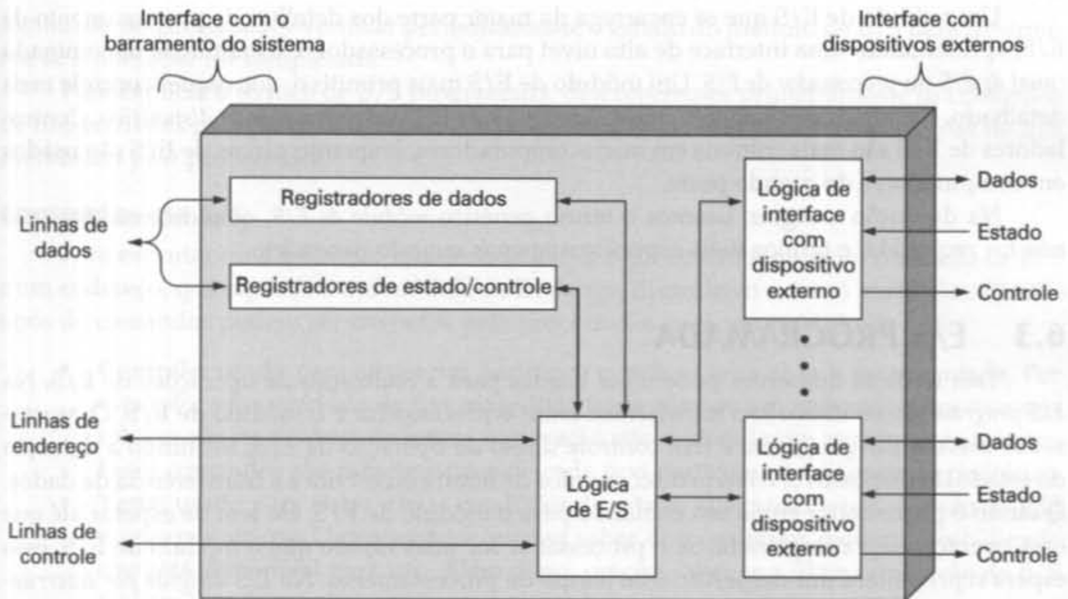


Figura 6.4 Diagrama de blocos de um módulo de E/S.

Estrutura do módulo de E/S

A complexidade de um módulo de E/S e o número de dispositivos externos que ele controla variam consideravelmente. Uma descrição bastante geral é apresentada a seguir (a Seção 6.4 descreve um dispositivo específico, o Intel 82C55A). A Figura 6.4 mostra um diagrama de blocos de um módulo de E/S genérico. O módulo é conectado ao restante do computador por meio de um conjunto de linhas de sinal (por exemplo, as linhas do barramento do sistema). Os dados transferidos desse módulo e para esse módulo são armazenados temporariamente em um ou mais registradores de dados. Pode também haver um ou mais registradores de estado, que fornecem dados sobre o estado corrente. Um registrador de estado pode também funcionar como registrador de controle, recebendo do processador informação de controle detalhada. A lógica interna do módulo interage com o processador por meio de um conjunto de linhas de controle. Essas linhas são usadas pelo processador para enviar comandos para o módulo de E/S. Algumas linhas de controle podem ser usadas pelo módulo de E/S (por exemplo, para sinais de arbitração e de estado). O módulo de E/S deve também ser capaz de reconhecer e gerar endereços associados aos dispositivos que ele controla. Cada módulo de E/S tem um endereço distinto ou, caso ele controle mais de um dispositivo externo, um conjunto de endereços distintos. Finalmente, o módulo de E/S contém um circuito lógico específico para a interface de cada dispositivo que ele controla.

O módulo de E/S tem como função fornecer ao processador uma visão simplificada de uma ampla gama de dispositivos. Existe um grande espectro de capacidades que deve ser fornecido. Ele pode esconder detalhes de temporização, de formatos e da operação eletromecânica de um dispositivo externo, permitindo ao processador operar em termos de comandos de leitura e escrita simples e, possivelmente, de comandos para abrir e fechar arquivos. Um módulo de E/S, na sua forma mais simples, pode ainda deixar visível para o processador grande parte do trabalho do controle de dispositivos (por exemplo, rebobinar uma fita).

Um módulo de E/S que se encarrega da maior parte dos detalhes de processamento de E/S, apresentando uma interface de alto nível para o processador, é usualmente denominado *canal de E/S* ou *processador de E/S*. Um módulo de E/S mais primitivo, que requer controle mais detalhado, é normalmente denominado *controlador de E/S* ou *controlador de dispositivo*. Controladores de E/S são mais comuns em microcomputadores, enquanto canais de E/S são usados em computadores de grande porte.

Na descrição a seguir, usamos o termo genérico *módulo de E/S*, quando essa distinção não for requerida, e termos mais específicos apenas quando necessário.

6.3 E/S PROGRAMADA

Três técnicas diferentes podem ser usadas para a realização de operações de E/S. Na *E/S programada*, os dados são transferidos entre o processador e o módulo de E/S. O processador executa um programa e tem controle direto da operação de E/S, incluindo a detecção do estado do dispositivo, o envio de comandos de leitura ou escrita e a transferência de dados. Quando o processador envia um comando para o módulo de E/S, ele tem de esperar até que essa operação seja completada. Se o processador for mais rápido que o módulo de E/S, essa espera representará um desperdício de tempo de processamento. Na *E/S dirigida por interrupção*, o processador envia um comando de E/S e continua a executar outras instruções, sendo interrompido pelo módulo de E/S quando este tiver completado seu trabalho. Tanto na E/S programada quanto na E/S dirigida por interrupção, o processador é responsável por obter dados da memória principal, em uma operação de saída, e por armazenar dados na memória principal, em uma operação de entrada. A técnica alternativa é conhecida como *acesso direto à memória* (*direct memory access* — DMA). Nesse caso, a transferência de dados entre o módulo de E/S e a memória principal é feita diretamente sem envolver o processador.

A Tabela 6.3 relaciona essas três técnicas. A técnica de E/S programada é abordada nessa seção a seguir. As técnicas de E/S dirigida por interrupção e de DMA são discutidas nas duas seções seguintes, respectivamente.

Tabela 6.3 Técnicas de E/S

	Sem interrupções	**Com interrupções**
Transferência entre memória e E/S por meio do processador	E/S programada	E/S dirigida por interrupção
Transferência direta entre memória e E/S		Acesso direto à memória (DMA)

Visão geral

Quando um programa está sendo executado pelo processador, a execução de uma instrução relacionada a E/S faz com que um comando seja enviado para o módulo de E/S apropriado. Na E/S programada, o módulo de E/S executa a operação requisitada e sinaliza o término da operação carregando um valor apropriado no registrador de estado de E/S (Figura 6.4). Nenhuma outra ação é executada pelo módulo de E/S para alertar o processador sobre o término da operação. Em particular, o processador não é interrompido. Portanto, é respon-

sabilidade do processador verificar periodicamente o estado do módulo de E/S para determinar se a operação foi completada.

Para explicar a técnica de E/S programada, descreveremos primeiramente os comandos de E/S enviados pelo processador para o módulo de E/S e, em seguida, as instruções de E/S executadas pelo processador.

Comandos de E/S

Para executar uma instrução relacionada a E/S, o processador gera um comando de E/S e um endereço, que especifica um módulo de E/S e um dispositivo externo particular. Quatro tipos de comandos podem ser enviados pelo processador para um módulo de E/S:

- **Controle:** usado para ativar um periférico e indicar uma ação a ser executada. Por exemplo, uma unidade de fita magnética pode receber um comando para que seja rebobinada ou a cabeça de leitura e gravação seja deslocada um registro para a frente. Esses comandos são característicos de cada tipo particular de dispositivo periférico.
- **Teste:** usado para testar várias condições de estado associadas a um módulo de E/S e seus periféricos. O processador precisa saber se o periférico requerido está ligado e se está disponível para uso. Além disso, precisa saber se a última operação de E/S foi completada e se houve algum erro.
- **Leitura:** faz com que o módulo de E/S obtenha um item de dado do periférico e o armazene em uma área de armazenamento temporário interna (representada, na Figura 6.4, como um registrador de dados). O processador poderá então obter esse item de dado solicitando ao módulo de E/S que o coloque no barramento de dados.
- **Gravação:** faz com que o módulo de E/S obtenha um item de dado (byte ou palavra) do barramento de dados e, em seguida, o transmita para o periférico.

A Figura 6.5a mostra um exemplo do uso de E/S programada para ler um bloco de dados (por exemplo, um registro de uma fita) de um dispositivo periférico para a memória. Uma palavra (por exemplo, 16 bits) é lida de cada vez. Para cada palavra lida, o processador permanece em um ciclo de verificação de estado até que se determine que uma palavra está disponível no registrador de dados do módulo de E/S. O fluxograma apresentado na figura assinala a principal desvantagem dessa técnica: é um processo que consome tempo do processador, mantendo-o desnecessariamente ocupado.

Instruções de E/S

Na E/S programada, há uma estreita correspondência entre as instruções de E/S que o processador busca na memória e os comandos de E/S que ele envia para o módulo de E/S, para a execução dessas instruções. Ou seja, as instruções são facilmente mapeadas em comandos de E/S e, freqüentemente, temos uma simples relação de um para um. A forma das instruções depende da maneira como os dispositivos externos são endereçados.

Tipicamente, há diversos dispositivos de E/S conectados ao sistema por meio de módulos de E/S. A cada dispositivo é associado um identificador ou endereço distinto. Quando o processador envia um comando de E/S, esse comando contém o endereço do dispositivo desejado. Dessa maneira, cada módulo de E/S deve interpretar as linhas de endereço para determinar se o comando lhe é destinado.

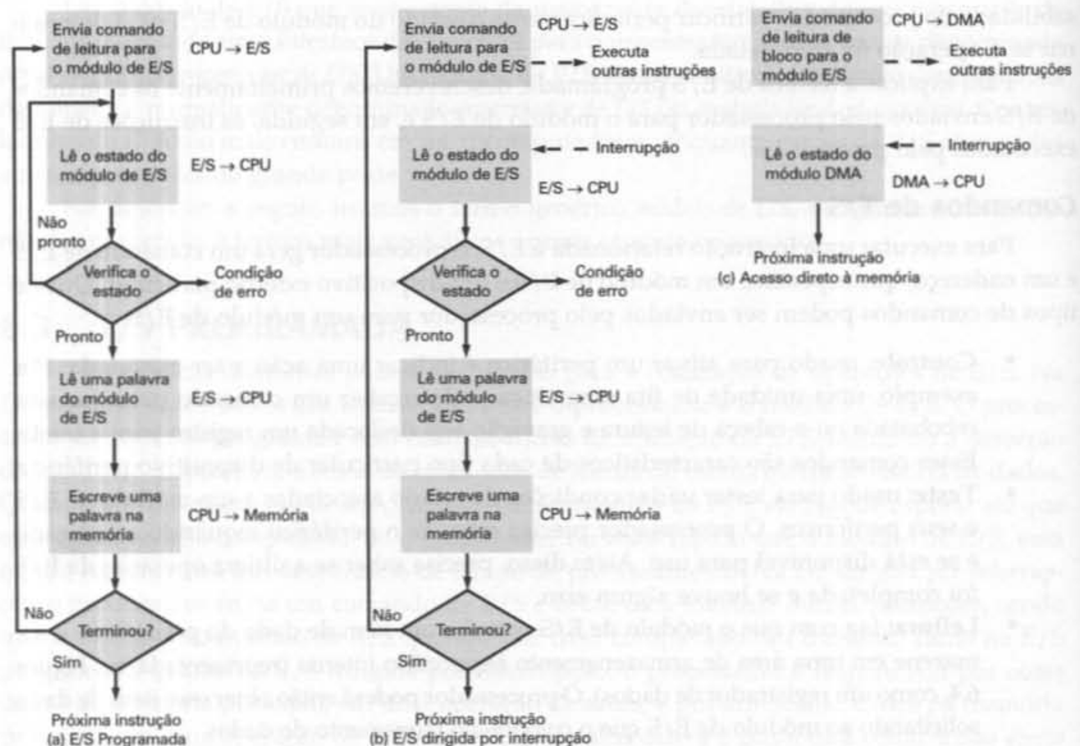
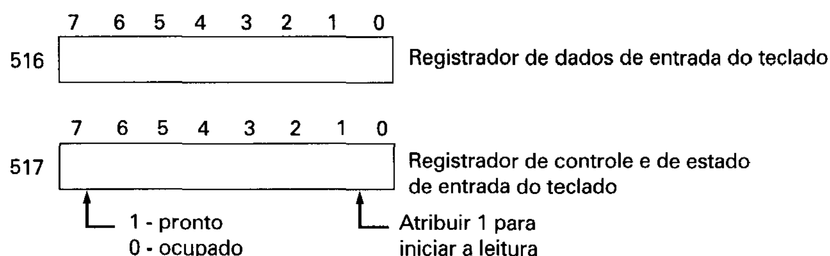


Figura 6.5 Três técnicas para a entrada de um bloco de dados.

Quando o processador, a memória principal e os módulos de E/S compartilham um barramento comum, dois modos de endereçamento diferentes podem ser usados: endereçamento mapeado na memória e endereçamento independente. Na *E/S mapeada na memória*, existe um único espaço de endereçamento para posições de memória e dispositivos de E/S. Os registradores de dados e de estado dos módulos de E/S são vistos pelo processador como posições de memória, e as mesmas instruções de máquina são usadas para acessar a memória ou os dispositivos de E/S. Por exemplo, com dez linhas de endereço, é possível obter uma combinação com um total de $2^{10} = 1024$ endereços de memória e de dispositivos de E/S.

Na E/S mapeada na memória, o barramento precisa ter apenas uma linha para leitura e uma linha para escrita. Alternativamente, ele pode incluir, além das linhas de leitura e escrita, linhas de comando de entrada e saída. Nesse caso, uma linha de comando especifica se um endereço corresponde a uma posição de memória ou a um dispositivo de E/S. Qualquer endereço pode corresponder a uma posição de memória ou a um dispositivo de E/S. Com dez linhas de endereço, o sistema pode incluir 1024 posições de memória e 1024 endereços de E/S. Como o espaço de endereçamento de E/S é independente do espaço de endereçamento de memória, esse modo é denominado *E/S independente*.



ENDEREÇO	INSTRUÇÃO	OPERANDO	COMENTÁRIO
200	Carregar acumulador	"1"	
	Armazenar acumulador	517	Iniciar leitura do teclado
202	Carregar acumulador	517	Obter byte de estado
	Desviar se sinal = 0	202	Repetir até que esteja pronto
	Carregar acumulador	516	Carregar byte de dados

(a) E/S mapeada na memória

ENDEREÇO	INSTRUÇÃO	OPERANDO	COMENTÁRIO
200	Iniciar E/S	5	Iniciar leitura do teclado
201	Testar E/S	5	Testar se a operação foi completada
	Desviar se não pronto	201	Repetir até que seja completada
	Leitura	5	Carregar byte de dados

(b) E/S independente

Figura 6.6 E/S mapeada na memória e E/S independente.

A Figura 6.6 compara essas duas técnicas de E/S programada. A Figura 6.6a mostra como pode ser a interface do programador para um dispositivo de entrada simples; tal como um teclado, pode aparecer para um programador usando E/S mapeada na memória. Considere um endereço de 10 bits, uma memória com 512 bytes (endereços 0-511) e até 512 endereços de E/S (endereços 512-1023). Dois desses endereços (516 e 517) são alocados para a entrada de dados a partir do teclado de um terminal particular. O endereço 516 corresponde ao registrador de dados e o endereço 517, ao registrador de estado, que também funciona como registrador de controle para a recepção de comandos do processador. O programa mostrado lê 1 byte de dados do teclado e armazena o valor lido no registrador AC (acumulador) do processador. Note que o processador permanece em um laço de teste até que o byte de dados esteja disponível.

Na E/S independente (Figura 6.6b), o acesso às portas de E/S é feito por meio de comandos de E/S especiais, que ativam as linhas de comando de E/S do barramento.

Na maioria dos tipos de processador, há um conjunto relativamente grande de instruções que se referem à memória. Na E/S independente, existe apenas um pequeno número de instruções de E/S. Dessa maneira, o fato de o repertório grande de instruções existentes para acessar a memória poder também ser usado para E/S é uma vantagem da E/S mapeada na memória, possibilitando uma programação mais eficiente. Uma desvantagem é que alguns

endereços não podem ser usados para referência à memória, sendo associados a dispositivos de E/S. Tanto a E/S mapeada na memória quanto a E/S independente são comuns.

6.4 E/S DIRIGIDA POR INTERRUPÇÃO

O problema da E/S programada é que o processador tem de esperar um longo tempo até que o módulo de E/S requerido esteja pronto para receber ou enviar dados. Enquanto espera, o processador tem de testar, continuamente, o estado do módulo de E/S. Como resultado, o desempenho global do sistema é severamente degradado.

Uma alternativa é o processador enviar um comando de E/S para um módulo e continuar a executar outras instruções. O processador será interrompido pelo módulo de E/S quando este estiver pronto para trocar dados com o processador. O processador efetua então a transferência de dados, como na técnica anterior, e depois retoma o seu processamento original.

Vejamos como isso funciona, primeiramente do ponto de vista do módulo de E/S. Em uma entrada de dados, o módulo de E/S recebe um comando READ (leitura) do processador. Então, lê o dado requerido do periférico especificado. Quando esse dado estiver em seu registrador de dados, o módulo de E/S sinaliza a ocorrência de uma interrupção do processador por meio de uma linha de controle. Ele então espera até que o dado lido seja solicitado pelo processador. Quando recebe essa requisição, ele coloca esse dado no barramento de dados e fica preparado para uma nova operação de E/S.

Do ponto de vista do processador, a entrada de dados é executada do seguinte modo. O processador envia um comando READ para o módulo de E/S e prossegue com a execução de outras instruções (por exemplo, de outros programas que estejam sendo executados simultaneamente). No final de cada ciclo de instrução, ele verifica se existe alguma interrupção pendente (Figura 3.9). Quando detecta uma interrupção de E/S, ele salva o contexto do programa corrente (por exemplo, o contador de programa e demais registradores) e processa a interrupção, lendo a palavra de dados do módulo de E/S e armazenando-a na memória. Restaura então o contexto do programa que foi interrompido (ou de algum outro programa) e reinicia sua execução.

A Figura 6.5b mostra o uso de E/S dirigida por interrupção para a leitura de um bloco de dados. Compare-a com a Figura 6.5a. A E/S dirigida por interrupção é mais eficiente que a E/S programada, pois elimina ciclos de espera desnecessários. No entanto, ela ainda consome muito tempo do processador, pois cada palavra de dados transferida do módulo de E/S para a memória, ou vice-versa, tem de passar pelo processador.

Processamento de interrupção

Examinemos mais detalhadamente o papel do processador na E/S dirigida por interrupção. A ocorrência de uma interrupção dispara certo número de eventos, tanto no hardware quanto no software do processador. A Figura 6.7 mostra uma seqüência de eventos típica. Quando um dispositivo completa uma operação de E/S, ocorre a seguinte seqüência de eventos de hardware:

1. O dispositivo envia um sinal de interrupção para o processador.
2. Antes de responder a essa interrupção, o processador termina a execução da instrução corrente, como indicado na Figura 3.9.

3. O processador testa se existe uma interrupção pendente e, quando detectada, envia um sinal de reconhecimento para o dispositivo que enviou a interrupção. O recebimento desse sinal faz com que o dispositivo desative seu sinal de interrupção.
4. O processador agora se prepara para transferir o controle para a rotina de tratamento da interrupção. Primeiramente, ele armazena, para posterior recuperação, os dados necessários para retomar a execução do programa corrente a partir do ponto em que foi interrompida. A informação mínima necessária consiste (a) no estado do processador, contido em um registrador denominado palavra de estado de programa (*program status word* — PSW), e (b) no endereço da próxima instrução a ser executada, contido no contador de programa (*program counter* — PC). Essas informações podem ser armazenadas na pilha de controle do sistema[2].

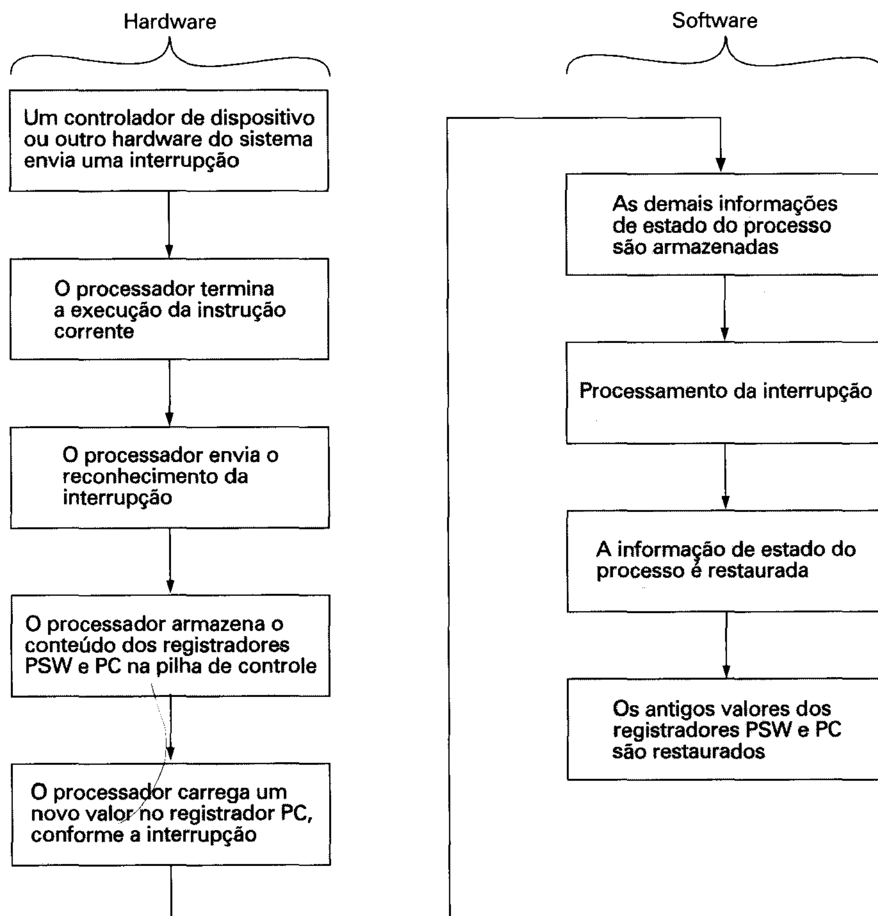


Figura 6.7 Processamento de interrupção simples.

2. Veja o Apêndice 9A para uma descrição da operação de uma pilha.

5. O processador carrega então o contador de programa com o endereço da rotina de tratamento de interrupção. Dependendo da arquitetura do computador e do projeto do sistema operacional, pode haver uma única rotina de tratamento de interrupção, ou uma rotina para cada tipo de interrupção, ou uma rotina para cada dispositivo e cada tipo de interrupção. Se existir mais de uma rotina de tratamento de interrupção, o processador deve determinar qual deve ser chamada. Essa informação pode ser enviada junto com o sinal de interrupção original ou pode ser obtida pelo envio de uma requisição para o dispositivo que enviou a interrupção, que então responde com a informação necessária.

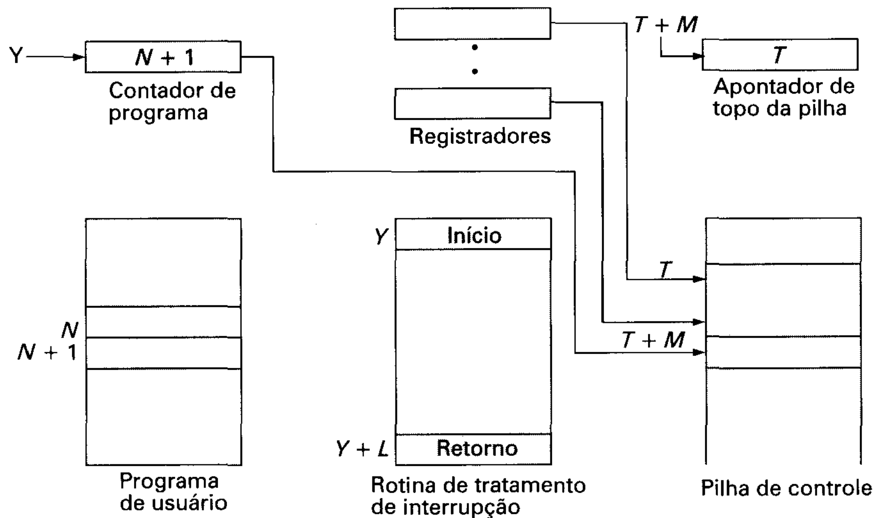
Uma vez que o contador de programa tenha sido carregado com o endereço da rotina de tratamento de interrupção, o processador reinicia o ciclo de execução de instruções, buscando a próxima instrução a ser executada. Como a busca da instrução é baseada no conteúdo do contador de programa, o controle é transferido para a rotina de tratamento de interrupção. A execução dessa rotina resulta nas seguintes operações:

6. Nesse ponto, os valores do contador de programa e do PSW do programa interrompido já terão sido devidamente armazenados na pilha do sistema para posterior restauração. Entretanto, outras informações que fazem parte do 'estado' do programa em execução também precisam ser guardadas. Em particular, o conteúdo dos registradores do processador, uma vez que esses registradores podem ser usados pela rotina de tratamento de interrupção. Assim todos esses valores, além de outras informações de estado, precisam ser salvos. Tipicamente, a rotina de tratamento de interrupção começa armazenando o conteúdo de todos os registradores na pilha do sistema, como ilustrado na Figura 6.8a. Nessa figura, o programa de usuário é interrompido após a execução da instrução de endereço N. O conteúdo de cada registrador, assim como o endereço da próxima instrução ($N + 1$) são armazenados na pilha. O apontador da pilha é atualizado para apontar para o novo topo da pilha, e o contador de programa é carregado com o endereço de início da rotina de tratamento de interrupção.
7. A rotina de tratamento de interrupção é então iniciada. O tratamento de interrupção inclui a verificação de informações de estado relacionadas à operação de E/S ou a outro evento que tenha causado a interrupção. Isso pode também envolver o envio de comandos adicionais ou de sinais de reconhecimento para o dispositivo de E/S.
8. Quando o processamento da interrupção é concluído, os valores anteriormente armazenados na pilha são restaurados nos registradores (por exemplo, veja a Figura 6.8b).
9. A última operação consiste em restaurar os conteúdos do PSW e do contador de programa. Com isso, a próxima instrução executada será uma instrução do programa previamente interrompido.

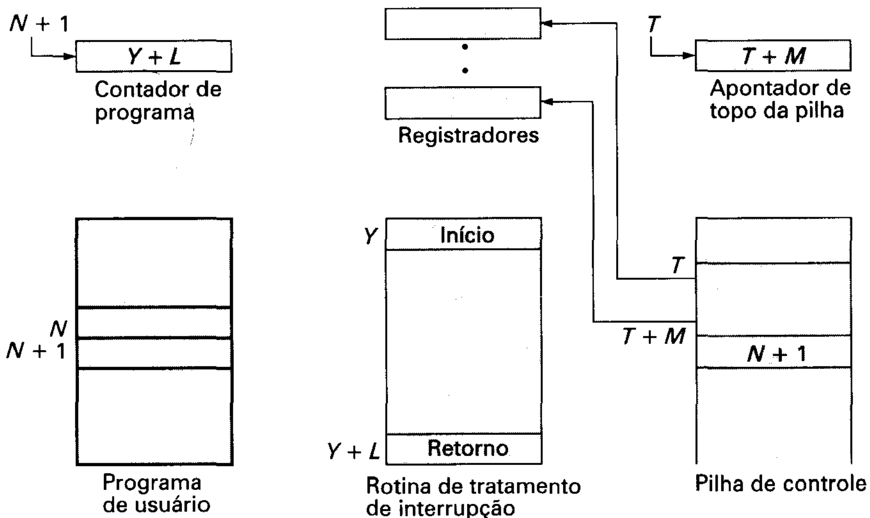
Note que é importante que todos os dados que caracterizam o estado do programa sejam salvos, para que seja possível retomar a execução desse programa posteriormente. Isso porque uma interrupção não é uma rotina chamada pelo próprio programa. Uma interrupção pode ocorrer a qualquer instante e, portanto, em qualquer ponto da execução de um programa de usuário. Sua ocorrência é imprevisível. Como veremos no próximo capítulo, o programa que solicitou a operação de E/S ocasionando a interrupção pode não ter nada em comum com o programa que é interrompido, podendo até pertencer a um usuário diferente.

Aspectos de projeto

Duas questões de projeto devem ser consideradas na implementação de E/S dirigida por interrupção: primeira, como o processador determina, entre os vários módulos de E/S existentes, qual dispositivo enviou a interrupção, e segunda, quando ocorrerem várias interrupções, como o processador decide qual ele deve processar.



(a) A interrupção ocorre após a execução da instrução de endereço *N*



(b) Retorno da interrupção

Figura 6.8 Alterações na memória e nos registradores durante o tratamento de uma interrupção.

Consideramos primeiramente a identificação do dispositivo. Quatro tipos de técnicas são mais usadas para identificar o dispositivo que enviou uma interrupção:

- Múltiplas linhas de interrupção
- Identificação por software
- *Daisy chain* (identificação por hardware, vetorada)
- Arbitração do barramento (vetorada)

A abordagem mais direta para o problema da identificação do dispositivo originador da interrupção consiste em usar *múltiplas linhas de interrupção* entre o processador e os módulos de E/S. Entretanto, na prática, apenas um pequeno número de linhas do barramento, ou pinos do processador, pode ser usado para as linhas de interrupção. Por isso, mesmo nesse caso, é bem provável que diversos módulos de E/S sejam conectados a uma mesma linha. Portanto, uma das três técnicas a seguir terá de ser usada, em cada linha.

Uma técnica alternativa é a *identificação por software*. Quando o processador detecta uma interrupção pendente, ele desvia a execução para uma rotina de tratamento de interrupção que interroga cada módulo de E/S para determinar qual deles causou a interrupção. Uma linha de comando especial (por exemplo, TEST I/O) pode ser usada para isso. Nesse caso, o processador ativa o sinal na linha TEST I/O e coloca o endereço de um determinado módulo de E/S nas linhas de endereço. O módulo de E/S correspondente responde afirmativamente caso tenha enviado a interrupção. Alternativamente, cada módulo pode conter um registrador de estado endereçável, que é lido pelo processador para identificar o módulo de E/S que causou a interrupção. Uma vez que o módulo tenha sido identificado, o processador chama a rotina de tratamento de interrupção específica para esse dispositivo.

A desvantagem da identificação por software é que consome muito tempo. Uma técnica mais eficiente é usar um *daisy chain*, que realiza a identificação por hardware, usando uma conexão entre os módulos e o processador, na forma de uma cadeia circular. Um exemplo de uma configuração desse tipo é mostrado na Figura 3.25. Todos os módulos de E/S compartilham uma linha de requisição de interrupção comum. A linha de reconhecimento de interrupção é estruturada em uma cadeia circular. Quando o processador recebe um sinal de interrupção, ele envia um sinal de reconhecimento de interrupção, que se propaga por meio de uma série de módulos de E/S até chegar àquele que causou a interrupção. Esse módulo então responde colocando uma palavra nas linhas de dados. Essa palavra é denominada *vetor de interrupção* e consiste no endereço do módulo de E/S ou algum outro tipo de identificador do módulo. O vetor de interrupção é usado pelo processador para determinar a rotina de tratamento de interrupção apropriada para aquele dispositivo. Isso evita a execução inicial de uma rotina genérica de tratamento de interrupção, que apenas determina o endereço da rotina específica. Essa técnica é conhecida como *interrupção vetorada*.

Outra técnica que usa interrupções vetoradas é a *arbitração do barramento*. Para enviar um sinal de interrupção, o módulo de E/S precisa primeiramente obter o controle do barramento. Dessa maneira, apenas um módulo de E/S pode ativar a linha de interrupção de cada vez. Quando o processador detecta a interrupção, ele responde por meio da linha de reconhecimento de interrupção. Então, o módulo que causou a interrupção coloca seu vetor nas linhas de dados.

As técnicas descritas anteriormente servem para identificar o módulo de E/S que causou uma interrupção, além de estabelecer prioridades para as interrupções pendentes. Quan-

do diversas linhas de interrupção são usadas, o processador seleciona a linha de maior prioridade. Na técnica de identificação por software, a ordem em que os módulos são interrogados determina suas prioridades. Da mesma maneira, a ordem dos módulos na conexão em cadeia circular, usada na técnica *daisy chain*, determina suas prioridades. Finalmente, na técnica de arbitração do barramento, pode ser usado um esquema de prioridades tal como discutido na Seção 3.4.

Examinamos agora dois exemplos de estruturas de interrupção.

O controlador de interrupções Intel 82C59A

O Intel 80386 oferece uma única linha de requisição de interrupção (*interrupt request* — INTR) e uma única linha de reconhecimento de interrupção (*interrupt acknowledge* — INTA). Para que o 80386 possa manipular vários dispositivos e estruturas de prioridades, ele é normalmente configurado com um arbitrador de interrupção externo, o 82C59A. Os dispositivos externos são conectados ao 82C59A, que, por sua, vez é conectado ao 80386.

A Figura 6.9 mostra o uso do 82C59A para conectar vários módulos de E/S ao 80386. Um único 82C59A pode controlar até oito módulos. Caso seja necessário controlar mais de oito módulos, pode ser usado um arranjo em cascata, que possibilita controlar até 64 módulos.

A única responsabilidade do 82C59A é o gerenciamento de interrupções. O 82C59A recebe requisições de interrupção dos módulos a ele conectados, determina aquela que tem maior prioridade e então envia um sinal para o processador, na linha INTR. O processador responde com um sinal de reconhecimento, ativando a linha INTA. Ao receber esse sinal, o 82C59A coloca o vetor de interrupções apropriado no barramento de dados. O processador pode então processar a interrupção, comunicando-se diretamente com o módulo de E/S, para ler ou escrever dados.

O 82C59A é programável. O 80386 especifica o esquema de prioridades a ser usado, carregando um valor na palavra de controle do 82C59A. Os seguintes modos de operação são possíveis:

- **Totalmente aninhado:** as requisições de interrupção são ordenadas de acordo com as prioridades de 0 (IR0) a 7 (IR7).
- **Circular:** em algumas aplicações, diversos dispositivos possuem a mesma prioridade de interrupção. Nesse modo de operação, quando a interrupção de um dispositivo acaba de ser atendida, o dispositivo recebe a prioridade mais baixa do grupo.
- **Máscara especial:** possibilita ao processador inibir interrupções de determinados dispositivos.

A interface de periféricos programável Intel 82C55A

Para exemplificar um módulo de E/S usado para E/S programada e para E/S dirigida por interrupção, consideramos a interface de periféricos programável Intel 82C55A. O 82C55A é um módulo de E/S de propósito geral, que consiste em uma única pastilha, projetada para ser usada junto com o processador Intel 80386. A Figura 6.10 mostra um diagrama de blocos geral do 82C55A, incluindo a designação dos 40 pinos do encapsulamento em que a pastilha é alojada.

O lado direito do diagrama de blocos é a interface externa do 82C55A. As 24 linhas de E/S podem ser programadas pelo 80386, por meio do registrador de controle do 82C55A e especificar uma configuração e um modo de operação. As 24 linhas são divididas em três gru-

pos de 8 bits (A, B e C). Cada grupo pode funcionar como uma porta de E/S de 8 bits. Além disso, o grupo C é subdividido em dois grupos de 4 bits (C_A e C_B), que podem ser usados junto com as portas A e B de E/S, para conter sinais de estado e de controle.

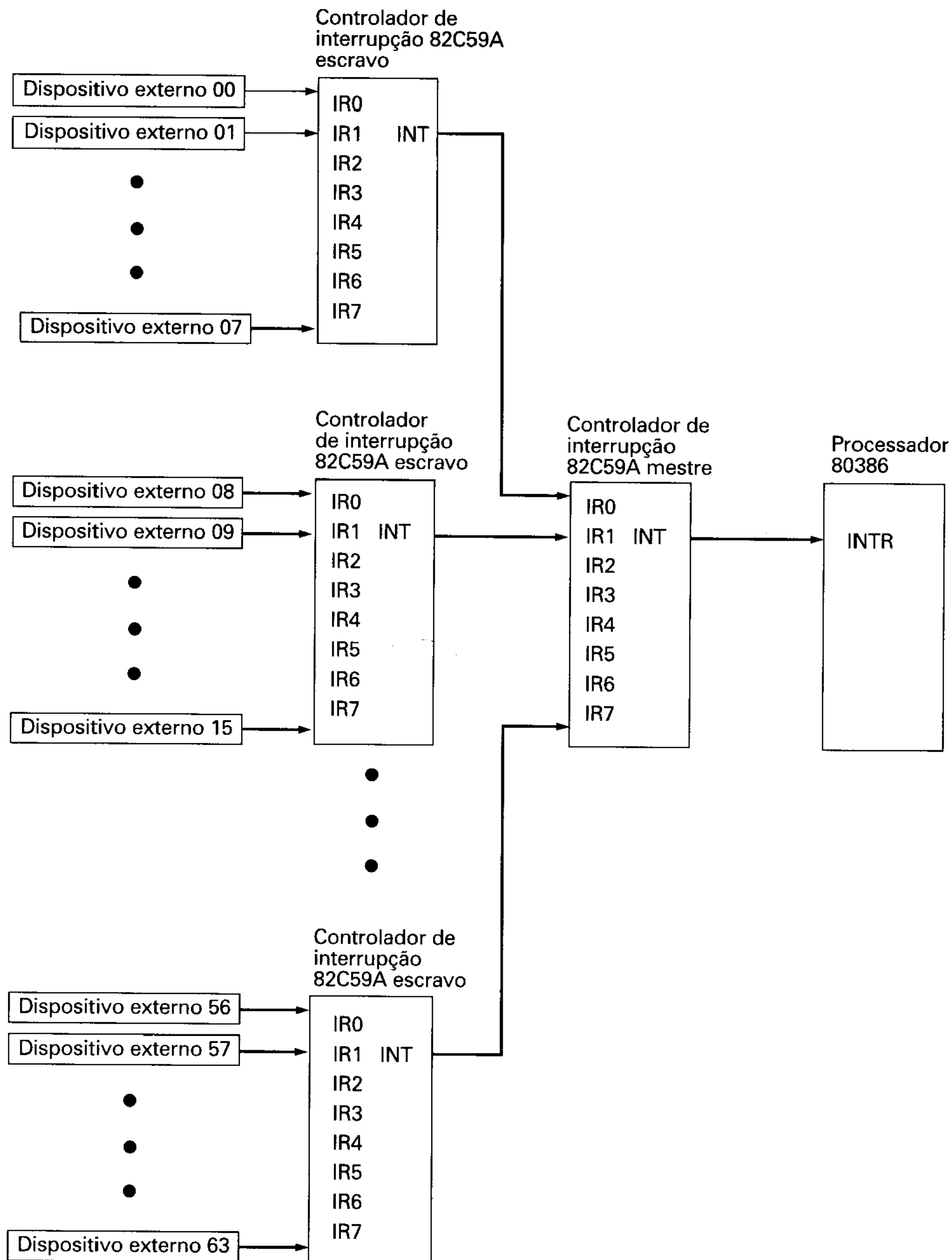


Figura 6.9 Uso do controlador de interrupção 82C59A.

O lado esquerdo do diagrama de blocos representa a interface interna com o barramento do 80386. Essa interface inclui um barramento de dados bidirecional de 8 bits (D0 a D7), usado para transferir dados de e para as portas de E/S, além de um valor para o registrador de controle. Duas linhas de endereço especificam uma das três portas de E/S ou o registrador de controle. Uma transferência ocorre quando a linha CHIP SELECT (seleção de pastilha) é habilitada, juntamente com a linha READ ou WRITE. A linha RESET é usada para inicializar a operação do módulo.

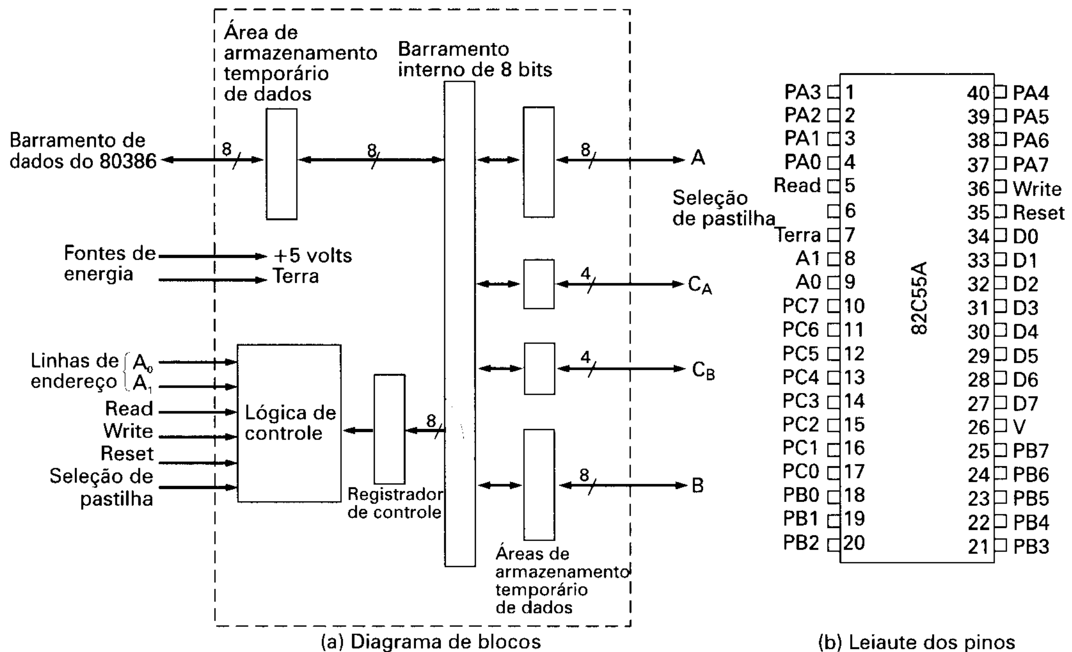


Figura 6.10 Diagrama de blocos da interface de periféricos programável Intel 82C55A.

O processador é responsável pela carga de um certo valor ao registrador de controle para determinar seu modo de operação e definir os sinais, se houver. No modo de operação 0, os três grupos de oito linhas externas funcionam como três portas de E/S de 8 bits. Cada porta pode ser designada como uma porta de entrada ou de saída. Nos demais modos, as linhas dos grupos *A* e *B* funcionam como portas de E/S e as do grupo *C*, como linhas de controle para os grupos *A* e *B*. Os sinais de controle têm duas funções principais: 'controle de comunicação' e requisição de interrupção. A função de controle de comunicação diz respeito à temporização das operações. Uma linha de controle é usada como uma linha DATA READY (dados prontos) pelo transmissor de uma operação para indicar a existência de dados nas linhas de dados de E/S. Outra linha de controle é usada como uma linha ACKNOWLEDGE pelo receptor para enviar um sinal de reconhecimento, indicando que os dados foram lidos e que os sinais nas linhas de dados podem ser desativados. Outra linha pode ser designada como uma linha INTERRUPT REQUEST usada para requisitar uma interrupção, sendo conectada diretamente ao barramento do sistema.

Como a interface 82C55A pode ser programada por meio do seu registrador de controle, ela pode ser usada para controlar uma variedade de dispositivos periféricos simples. A Figura 6.11 ilustra o uso do 82C55A para controlar um teclado/terminal de vídeo. O teclado fornece 8 bits de entrada, dois dos quais, SHIFT e CONTROL, possuem significado especial para o programa de controle de teclado sendo executado pelo processador. No entanto, a interpretação desses bits é transparente para o 82C55A, que simplesmente recebe os 8 bits de dados do teclado e os envia para o barramento de dados do sistema. Além das linhas de dados, são fornecidas duas linhas de controle para a comunicação com o teclado.

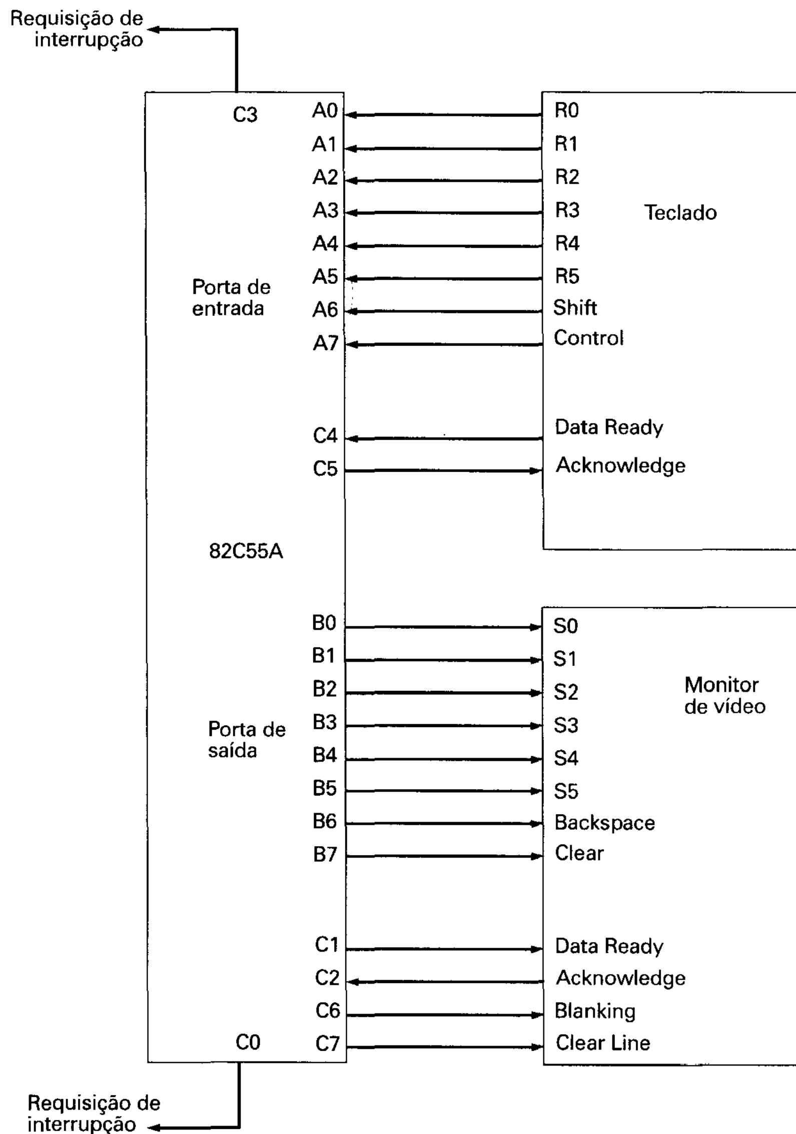


Figura 6.11 Interface de teclado/terminal de vídeo com o 82C55A.

O monitor de vídeo é também ligado ao 82C55A por meio de uma porta de dados de 8 bits. Mais uma vez, dois desses bits têm significado especial (BACKSPACE e CLEAR), transparente para o 82C55A. São também usadas duas linhas de controle para a comunicação com o monitor (DATA READY e ACKNOWLEDGE) e duas outras linhas para funções de controle adicionais (BLANKING e CLEAR LINE).

6.5 ACESSO DIRETO À MEMÓRIA (DMA)

Desvantagens da E/S programada e da E/S dirigida por interrupção

A E/S dirigida por interrupção, embora mais eficiente que a E/S programada, ainda requer uma intervenção ativa do processador para transferir dados entre a memória e o módulo de E/S, e toda transferência é feita por um caminho que passa pelo processador. Desse modo, essas duas formas de E/S possuem duas desvantagens inerentes:

1. A taxa de transferência de E/S é limitada pela velocidade com que o processador pode testar e servir um dispositivo.
2. O processador se ocupa de gerenciar a transferência de dados de E/S, tendo de executar várias instruções a cada transferência (veja a Figura 6.5).

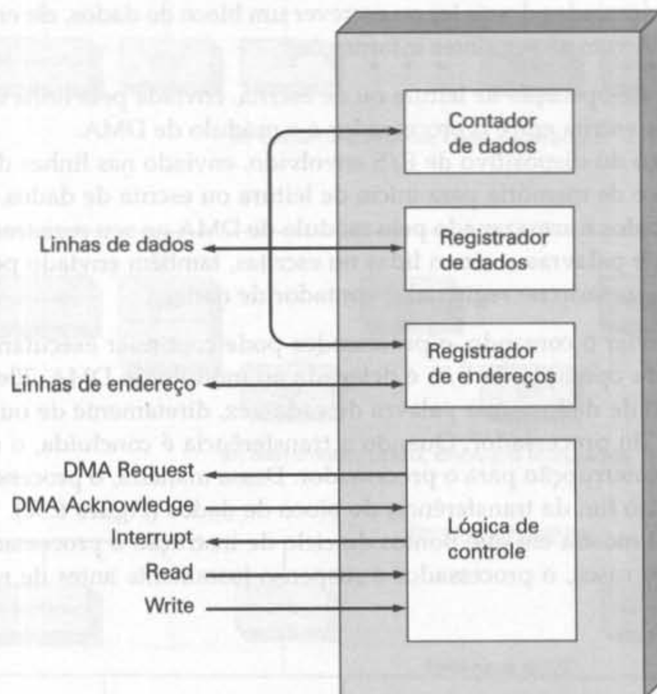


Figura 6.12 Diagrama de blocos de um módulo de DMA.

Essas duas desvantagens são, de certo modo, conflitantes. Considere a transferência de um bloco de dados. Na E/S programada, o processador é responsável pela E/S e pode transferir dados a uma taxa mais alta, à custa de não poder executar outras operações. Na E/S dirigida por interrupção, o processador fica, até certo ponto, liberado para executar outras operações, mas a preço de uma taxa de transferência de E/S mais baixa. Contudo, ambos os métodos têm um impacto adverso tanto na atividade do processador quanto na taxa de transferência de E/S.

Uma técnica mais eficiente para a transferência de grandes volumes de dados é a técnica de acesso direto à memória (DMA).

Funcionamento da E/S por acesso direto à memória (DMA)

A técnica de acesso direto à memória envolve um módulo adicional no barramento do sistema. Esse módulo ou controlador de DMA (Figura 6.12) é capaz de imitar o processador e, de fato, controlar o sistema do processador. Isso é necessário para que o módulo de DMA possa transferir dados diretamente de e para a memória por meio do barramento do sistema. Para esse propósito, um módulo de DMA pode tanto usar o barramento apenas quando este não está sendo usado pelo processador quanto forçar o processador a suspender sua operação temporariamente. Essa última técnica é mais comum, sendo conhecida como *roubo de ciclo*, porque o módulo de DMA de fato rouba um ciclo de barramento do processador.

Quando o processador deseja ler ou escrever um bloco de dados, ele envia um comando ao módulo de DMA com as seguintes informações:

- Indicação de operação de leitura ou de escrita, enviada pela linha de controle de leitura ou de escrita entre o processador e o módulo de DMA.
- O endereço do dispositivo de E/S envolvido, enviado nas linhas de dados.
- O endereço de memória para início de leitura ou escrita de dados, enviado pelas linhas de dados e armazenado pelo módulo de DMA no seu registrador de endereços.
- Número de palavras a serem lidas ou escritas, também enviado pelas linhas de dados e armazenado no registrador contador de dados.

Depois de enviar o comando, o processador pode continuar executando outras instruções. A execução da operação de E/S é delegada ao módulo de DMA. Ele transfere diretamente todo o bloco de dados, uma palavra de cada vez, diretamente de ou para a memória, sem a intervenção do processador. Quando a transferência é concluída, o módulo de DMA envia um sinal de interrupção para o processador. Dessa maneira, o processador é envolvido apenas no início e no fim da transferência do bloco de dados (Figura 6.5c).

A Figura 6.13 mostra em que pontos do ciclo de instrução o processador pode ser suspenso. Em todos os casos, o processador é suspenso justamente antes de requerer o uso do barramento.

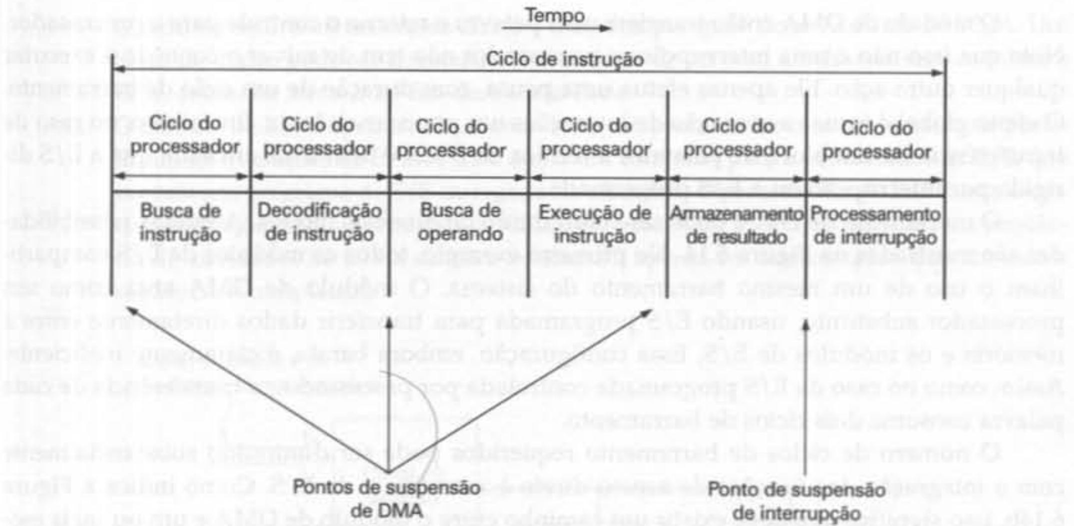
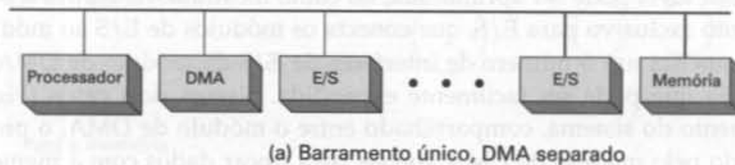
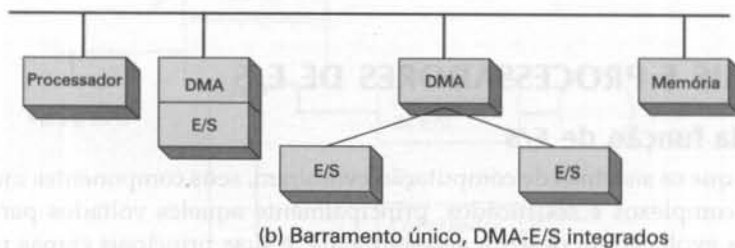


Figura 6.13 Pontos de suspensão de DMA e de interrupção ao longo de um ciclo de instrução.





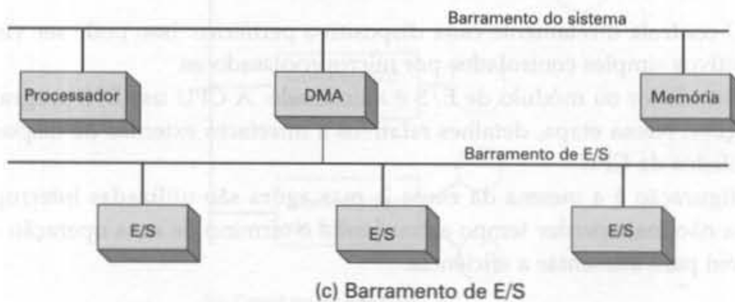


Figura 6.14 Configurações alternativas de DMA.

O módulo de DMA então transfere uma palavra e retorna o controle para o processador. Note que isso não é uma interrupção; o processador não tem de salvar o contexto e executar qualquer outra ação. Ele apenas efetua uma pausa, com duração de um ciclo de barramento. O efeito global é tornar a execução de instruções um pouco mais lenta. Entretanto, no caso de transferência de um bloco de palavras, a técnica de DMA é bem mais eficiente que a E/S dirigida por interrupção ou a E/S programada.

O mecanismo de DMA pode ser configurado de diversos modos. Algumas possibilidades são mostradas na Figura 6.14. No primeiro exemplo, todos os módulos de E/S compartilham o uso de um mesmo barramento do sistema. O módulo de DMA atua como um processador substituto, usando E/S programada para transferir dados diretamente entre a memória e os módulos de E/S. Essa configuração, embora barata, é claramente ineficiente. Assim como no caso da E/S programada controlada por processador, a transferência de cada palavra consome dois ciclos de barramento.

O número de ciclos de barramento requeridos pode ser diminuído substancialmente com a integração das funções de acesso direto à memória e de E/S. Como indica a Figura 6.14b, isso significa que deve existir um caminho entre o módulo de DMA e um ou mais módulos de E/S, independente do barramento do sistema. A lógica de controle do DMA pode ser parte de um módulo de E/S ou ser um módulo separado que controla um ou mais módulos de E/S. Essa idéia pode ser aprimorada, tal como mostrado na Figura 6.14c, com o uso de um barramento exclusivo para E/S, que conecta os módulos de E/S ao módulo de DMA. Isso reduz para apenas um o número de interfaces de E/S do módulo de DMA e possibilita uma configuração que pode ser facilmente expandida. Nesses dois casos (Figuras 6.14b e 6.14c), o barramento do sistema, compartilhado entre o módulo de DMA, o processador e a memória, é usado pelo módulo de DMA apenas para trocar dados com a memória. A transferência de dados entre o módulo de DMA e os módulos de E/S é feita fora do barramento do sistema.

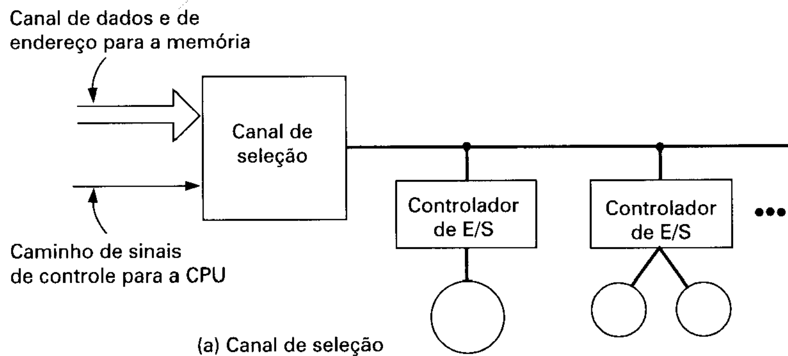
6.6 CANAIS E PROCESSADORES DE E/S

A evolução da função de E/S

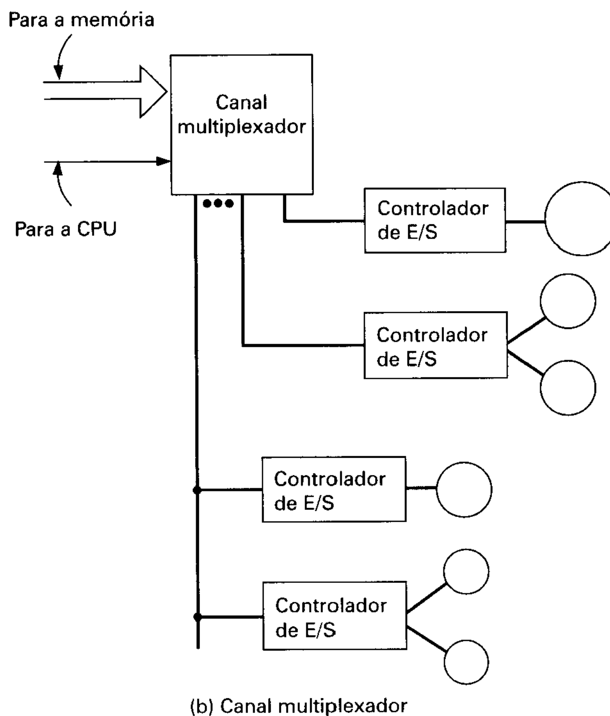
À medida que os sistemas de computação evoluíram, seus componentes individuais tornaram-se mais complexos e sofisticados, principalmente aqueles voltados para funções de E/S. Parte dessa evolução foi descrita anteriormente, e suas principais etapas podem ser resumidas como a seguir:

1. A CPU controla diretamente cada dispositivo periférico. Isso pode ser visto no caso de dispositivos simples controlados por microprocessadores.
2. Um controlador ou módulo de E/S é adicionado. A CPU usa E/S programada sem interrupções. Nessa etapa, detalhes relativos a interfaces externas de dispositivos de E/S são isolados da CPU.
3. A configuração é a mesma da etapa 2, mas agora são utilizadas interrupções. A CPU passa a não mais perder tempo aguardando o término de uma operação de E/S, o que contribui para aumentar a eficiência.

4. O módulo de E/S efetua acesso direto à memória por meio de um módulo de DMA. Um bloco de dados pode ser transferido diretamente de ou para a memória sem envolver a CPU, exceto no início e no fim da transferência.
5. O módulo de E/S é aprimorado, tornando-se um processador com um conjunto especializado de instruções de E/S. A CPU envia um comando para o processador de E/S, que executa um programa de E/S carregado na memória. O processador de E/S busca e executa instruções sem intervenção da CPU. Isso possibilita à CPU especificar uma seqüência de atividades de E/S a serem executadas e apenas ser interrompida quando toda a seqüência é completada.



(a) Canal de seleção



(b) Canal multiplexador

Figura 6.15 Arquiteturas de canais de E/S.

6. O módulo de E/S inclui uma memória local própria e é, portanto, ele próprio, um computador. Essa arquitetura possibilita controlar grande número de dispositivos de E/S, com o mínimo envolvimento da CPU. Ela normalmente é usada para controlar a comunicação com terminais interativos. O processador de E/S cuida da maior parte das tarefas envolvidas no controle dos terminais.

Ao longo desse caminho evolutivo, mais e mais funções de E/S são desempenhadas sem o envolvimento da CPU. A CPU é, cada vez mais, liberada de tarefas relacionadas a E/S, o que contribui para melhorar o desempenho global do sistema. Nas etapas 5 e 6 ocorre uma mudança importante com a introdução do conceito de um módulo de E/S capaz de executar um programa. O módulo de E/S da etapa 5 é freqüentemente denominado *canal de E/S*. Para o módulo da etapa 6, é mais usado o termo *processador de E/S*. Entretanto, esses dois termos são ocasionalmente utilizados em ambos os casos. Na descrição a seguir, usamos o termo *canal de E/S*.

Características dos canais de E/S

Um canal de E/S representa uma extensão do conceito de DMA. É capaz de executar instruções de E/S, o que lhe dá total controle sobre as operações de E/S. Em um sistema de computação que emprega esses dispositivos, a CPU não executa instruções de E/S. Essas instruções são armazenadas na memória principal, sendo executadas por um processador especializado no próprio canal de E/S. Dessa maneira, a CPU inicia uma transferência de E/S instruindo o canal de E/S para executar um determinado programa armazenado na memória. Esse programa especifica o dispositivo ou conjunto de dispositivos, a área ou as áreas da memória para armazenamento ou leitura de dados, a prioridade e as ações a serem tomadas no caso de determinadas situações de erro. O canal de E/S executa essas instruções e controla a transferência de dados.

Dois tipos de canais de E/S são comuns, como ilustrado na Figura 6.15. Um *canal seletor* controla vários dispositivos de E/S de alta velocidade e, a cada instante, fica dedicado para a transferência de dados de um desses dispositivos. Assim, o canal de E/S seleciona um dispositivo e efetua a transferência de dados. Cada dispositivo ou grupo de poucos dispositivos é conectado a um *controlador de E/S*, bastante semelhante aos módulos de E/S discutidos até agora. Dessa maneira, o canal de E/S controla esses controladores de E/S no lugar da CPU. Um *canal multiplexador* pode transferir dados para vários dispositivos ao mesmo tempo. Para dispositivos de E/S de baixa velocidade, é usado um *multiplexador de bytes*, que recebe ou transmite caracteres, o mais rápido possível, para diversos dispositivos. Por exemplo, caso três dispositivos estejam conectados, com taxas de transferência de dados diferentes, e cujos fluxos individuais de caracteres são A_1 A_2 A_3 A_4..., B_1 B_2 B_3 B_4... e C_1 C_2 C_3 C_4..., o fluxo de caracteres resultante pode ser A_1 B_1 C_1 A_2 C_2 A_3 B_2 C_3 A_4, e assim por diante. Para dispositivos de alta velocidade, é usado um *multiplexador de blocos*, que intercala a transferência de blocos de dados para os diversos dispositivos.

6.7 A INTERFACE EXTERNA: SCSI E *FireWire*

Tipos de interfaces

A interface de um módulo de E/S com um dispositivo periférico depende da natureza e da operação desse periférico. Uma característica importante é que uma interface pode ser serial ou paralela (Figura 6.16). Em uma *interface paralela,* existem várias linhas de conexão entre o módulo de E/S e o periférico e diversos bits são transferidos ao mesmo tempo, da mesma maneira como todos os bits de uma palavra são transferidos simultaneamente através do barramento de dados. Uma *interface serial* usa apenas uma linha para transmitir dados, sendo os bits transmitidos um de cada vez. A interface paralela é mais usada para periféricos de alta velocidade, tais como fitas e discos. A serial é mais comum para impressoras e terminais.

Em ambos os casos, o módulo de E/S tem de interagir com o periférico. Em termos gerais, a interação em uma operação de escrita pode ser descrita como a seguir:

1. O módulo de E/S envia um sinal de controle pedindo permissão para enviar um dado.
2. O periférico reconhece a requisição.
3. O módulo de E/S transfere dados (uma palavra ou um bloco, dependendo do tipo de periférico).
4. O periférico sinaliza o recebimento dos dados.

Uma operação de leitura é feita de modo semelhante.

Um ponto importante para a operação de um módulo de E/S é a utilização de uma área interna de armazenamento temporário, para manter dados que estão sendo transferidos entre o periférico e o restante do sistema. Isso permite ao módulo de E/S compensar diferenças de velocidade entre o barramento do sistema e suas linhas externas.

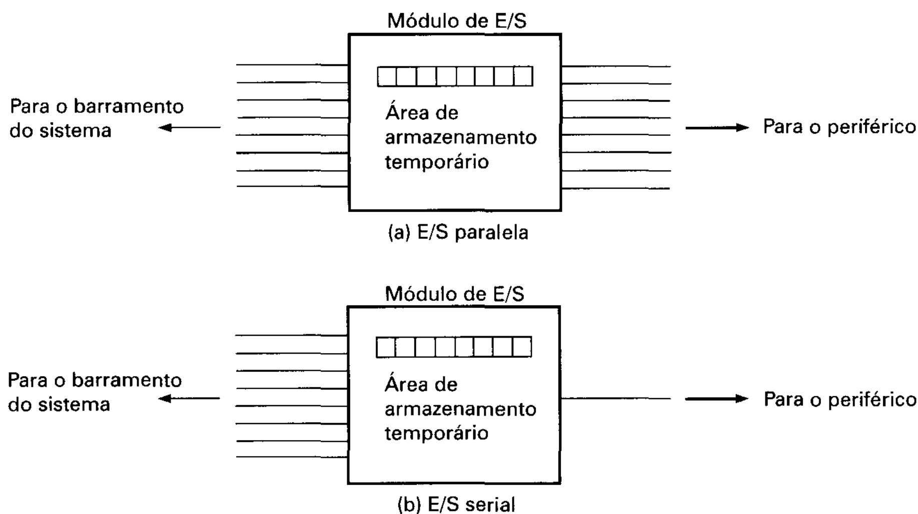


Figura 6.16 E/S paralela e serial.

Configurações ponto a ponto e multiponto

A conexão entre um módulo de E/S e os dispositivos externos pode ser ponto a ponto ou multiponto. Uma interface ponto a ponto oferece uma linha dedicada entre o módulo de E/S e o dispositivo externo. Em sistemas pequenos (tais como PCs e estações de trabalho), ligações ponto a ponto são usadas para a conexão com o teclado, impressoras e modem externo. Um exemplo desse tipo de interface é a especificação EIA-232. Uma descrição dessa especificação pode ser encontrada em Stallings (1997).

As interfaces externas multiponto estão se tornando cada vez mais importantes, sendo usadas para a conexão de dispositivos externos de armazenamento em massa (discos e fitas) e dispositivos de multimídia (CD-ROMs, vídeo, áudio). Elas são, de fato, barramentos externos que possuem o mesmo tipo de lógica dos barramentos discutidos no Capítulo 3. Dois exemplos importantes são examinados a seguir: SCSI e *FireWire*.

Interface de sistemas de computação pequenos (SCSI)

Um bom exemplo de interface para dispositivos periféricos externos é a SCSI. Popularizada inicialmente no Macintosh em 1984, a SCSI é empregada atualmente tanto no Macintosh quanto nos sistemas Windows/Intel, assim como em muitas estações de trabalho. Ela é uma interface padrão para unidades de CD-ROM, equipamentos de áudio e dispositivos externos de armazenamento em massa. Usa uma interface paralela com 8, 16 ou 32 linhas de dados.

A interface SCSI geralmente é referenciada como um barramento, embora os dispositivos sejam, de fato, conectados em uma cadeia circular. Cada dispositivo SCSI possui dois conectores: um para entrada e um para saída. Todos os dispositivos são conectados a uma cadeia circular, sendo um ponto dessa cadeia conectado ao computador. Eles funcionam de maneira independente e podem trocar dados entre si ou com o sistema hospedeiro. Por exemplo, o conteúdo de um disco rígido pode ser copiado diretamente para uma fita sem envolver o processador. Os dados são transferidos na forma de pacotes de mensagens, como descrito a seguir.

Versões da SCSI

A especificação SCSI original, atualmente denominada SCSI-1, foi desenvolvida no início da década de 80. A SCSI-1 contém oito linhas de dados e opera com velocidade de relógio de 5 MHz ou taxa de transferência de 5 Mbytes/s. Ela possibilita a conexão de até sete dispositivos ao sistema hospedeiro em uma cadeia circular.

Em 1991, foi introduzida uma revisão dessa especificação, a SCSI-2. As principais alterações foram a expansão opcional das linhas de dados para 16 ou 32 e o aumento da velocidade de relógio para 10 MHz. O resultado é uma taxa máxima de transferência de dados de 20 ou 40 Mbytes/s. Atualmente, está em andamento a especificação SCSI-3, que oferecerá velocidades ainda maiores.

Sinais e fases

Todas as transferências mediante o barramento SCSI ocorrem entre um *iniciador* e um *alvo*. Tipicamente, o sistema hospedeiro é o iniciador e um controlador de periférico, o alvo, mas alguns dispositivos podem assumir esses dois papéis. Toda atividade no barramento ocorre de acordo com uma seqüência de fases. As fases são as seguintes:

- **Barramento livre:** indica que nenhum dispositivo está usando o barramento e, portanto, ele está disponível para uso.
- **Arbitração:** permite a um dispositivo ganhar o controle do barramento, de maneira que possa iniciar ou retomar o processamento de uma E/S.
- **Seleção:** permite a um iniciador selecionar um alvo para realizar uma determinada função, tal como um comando de leitura ou de escrita.
- **Restabelecimento de conexão:** permite a um alvo restabelecer a conexão com um iniciador para retomar uma operação iniciada anteriormente, mas suspensa pelo alvo.
- **Comando:** permite ao alvo requisitar informação de comando ao iniciador.
- **Dados:** permite ao alvo requisitar a transferência de dados, do alvo para o iniciador (entrada de dados) ou do iniciador para o alvo (saída de dados).
- **Estado:** permite ao alvo realizar uma requisição para que uma informação de estado seja enviada ao iniciador.
- **Mensagem:** permite ao alvo requisitar a transferência de uma ou mais mensagens, do alvo para o iniciador (entrada de mensagem) ou do iniciador para o alvo (saída de mensagem).

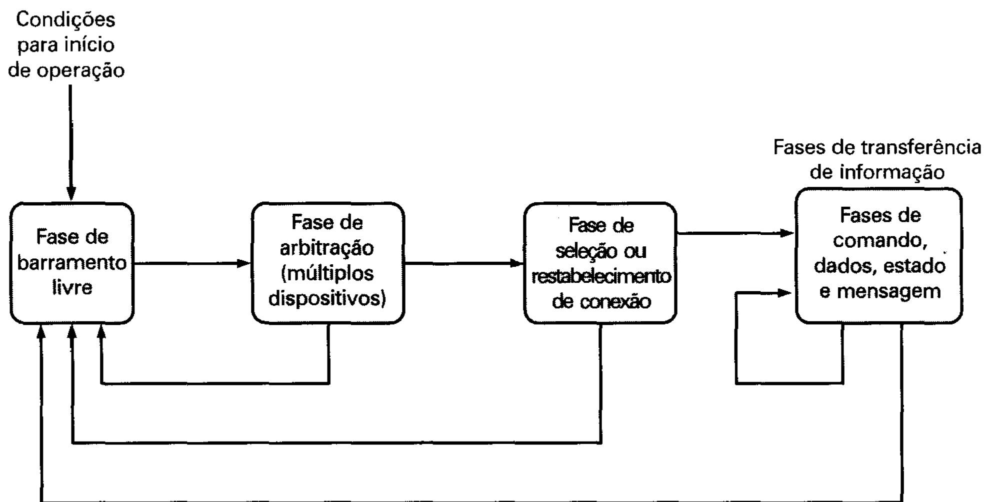


Figura 6.17 Fases do barramento SCSI.

A Figura 6.17 mostra a ordem em que essas fases do barramento SCSI ocorrem. Quando sua operação é iniciada (após o computador ser ligado ou reinicializado), o barramento entra na fase de *Barramento Livre*. Essa fase é seguida por uma fase de *Arbitração*, que normalmente resulta na aquisição do controle do barramento por um dispositivo. Caso a arbitração falhe, o barramento retorna à fase de *Barramento Livre*. Se a arbitração é bem-sucedida, o barramento entra na fase de *Seleção* ou de *Restabelecimento de Conexão*, que determina o dispositivo iniciador e o dispositivo-alvo para a transferência. Uma vez determinados o iniciador e o alvo, haverá uma ou mais fases de transferência de informação (*Comando*, *Dados*, *Estado* ou *Mensagem*) entre os dois dispositivos. A fase final de transferência de informação é, normalmente, uma fase de *Entrada de Mensagem*, em que é enviada uma mensagem de *Desconexão* ou de *Comando Completado* para o iniciador, sendo seguida pela fase de *Barramento Livre*.

Uma característica importante da SCSI é a capacidade de restabelecimento da conexão entre um iniciador e um alvo. Se a execução de um comando é demorada, o dispositivo-alvo pode liberar o barramento, podendo restabelecer a conexão com o iniciador mais tarde. Por exemplo, o processador pode enviar um comando de formatação a uma unidade de disco que efetua essa operação sem prender o barramento.

A especificação SCSI-1 define um cabo com 18 linhas de sinal, 9 linhas para controle e 9 linhas para dados (8 linhas de dados e 1 linha de paridade). As linhas de controle são descritas a seguir:

- **BSY:** usada por um iniciador ou alvo para indicar que o barramento está ocupado.
- **SEL:** utilizada por um iniciador para selecionar um alvo para executar um comando ou usada por um alvo para restabelecer conexão com seu iniciador, quando a conexão tiver sido suspensa. O sinal existente na linha **I/O** distingue esses dois casos (seleção e restabelecimento de conexão).

Tabela 6.4 Sinais do barramento SCSI

Fases do barramento	Sinais do cabo A					Sinais adicionais do cabo B		
	BSY	SEL	C/D, I/O, MSG, REQ	ACK, ATN	DB(7-0), DB(P)	REQP	ACKB	DB(31-8), DB(P1), DB(P2) DB(P3)
Barramento livre	Nenhum	Nenhum	Nenhum	Nenhum	Nenhum	Nenhum	Nenhum	Nenhum
Arbitração	Todos	Vencedor	Nenhum	Nenhum	ID S	Nenhum	Nenhum	Nenhum
Seleção	I&A	Iniciador	Nenhum	Iniciador	Iniciador	Nenhum	Nenhum	Nenhum
Restabelecimento de conexão	I&A	Alvo	Alvo	Iniciador	Alvo	Nenhum	Nenhum	Nenhum
Comando	Alvo	Nenhum	Alvo	Iniciador	Iniciador	Nenhum	Nenhum	Nenhum
Entrada de dados	Alvo	Nenhum	Alvo	Iniciador	Alvo	Alvo	Iniciador	Alvo
Saída de dados	Alvo	Nenhum	Alvo	Iniciador	Iniciador	Alvo	Iniciador	Iniciador
Estado	Alvo	Nenhum	Alvo	Iniciador	Alvo	Nenhum	Nenhum	Nenhum
Entrada de mensagem	Alvo	Nenhum	Alvo	Iniciador	Alvo	Nenhum	Nenhum	Nenhum
Saída de mensagem	Alvo	Nenhum	Alvo	Iniciador	Iniciador	Nenhum	Nenhum	Nenhum

Todos: o sinal é enviado por todos os dispositivos SCSI que competem pelo controle do barramento.

ID S: um único bit de dado (identificador SCSI), enviado por cada um dos dispositivos que competem pelo controle do barramento.

I&A: o sinal é enviado pelo iniciador, pelo alvo, ou por ambos, conforme especificado nas fases de *Seleção* e de *Restabelecimento de Conexão.*

Iniciador: enviado apenas pelo iniciador ativo.

Nenhum: o sinal não é enviado.

Vencedor: o sinal é enviado pelo dispositivo que obteve o controle do barramento.

Alvo: o sinal é enviado apenas pelo alvo ativo.

- **C/D:** usada pelo alvo para indicar se o barramento de dados contém informação de controle (comando, estado ou mensagem) ou dado.
- **I/O:** utilizada pelo alvo para indicar a direção de transferência de dados por meio do barramento de dados.
- **MSG:** usada pelo alvo para indicar ao iniciador que a informação que está sendo transferida é uma mensagem.
- **REQ:** empregada pelo alvo para requisitar uma transferência de dados. Em resposta ao sinal REQ, o iniciador lê o dado do barramento durante uma fase de *Entrada de Dados* ou coloca um dado no barramento durante uma fase de *Saída de Dados*.
- **ACK:** usada pelo iniciador para reconhecer um sinal REQ, enviado pelo alvo. O sinal ACK indica que o iniciador colocou uma informação no barramento em uma fase de *Saída de Dados* ou que o iniciador leu o dado do barramento em uma fase de *Entrada de Dados*.
- **ATN:** empregada pelo iniciador para informar ao alvo que ele tem uma mensagem para transferir. O iniciador envia esse sinal durante a fase de *Seleção* ou em qualquer momento depois que o alvo tenha assumido o controle do barramento.
- **RST:** usado para reinicializar a operação do barramento.

A Tabela 6.4 mostra a relação entre os sinais e as fases do barramento. Na especificação SCSI-2, existem linhas adicionais de dados e de paridade, além de mais um par de linhas REQ e ACK.

Temporização da SCSI

A Figura 6.18 mostra um diagrama de tempo típico da interface SCSI, que serve para explicar os diversos sinais e fases do barramento. O exemplo apresenta um comando de leitura que transfere dados do alvo para o iniciador.

A operação começa na fase de *Barramento Livre*, sem nenhum sinal em qualquer uma das linhas. Em seguida, ocorre uma fase de *Arbitração*, na qual um ou mais dispositivos competem pelo controle do barramento. Cada um desses dispositivos ativa a linha BSY e uma das linhas de dados. Cada um dos oito dispositivos (um hospedeiro e até sete outros dispositivos) possui um identificador (ID) distinto, de 0 a 7. Cada dispositivo ativa a linha de dados correspondente ao seu ID. Cada ID tem uma prioridade associada, onde 7 é a prioridade mais alta e 0 é a prioridade mais baixa. Durante a fase de *Arbitração*, se mais de um dispositivo ativar a linha de dados correspondente ao seu ID, o controle do barramento será cedido ao dispositivo cujo ID corresponde à linha de maior prioridade. Todos os dispositivos observam as linhas de dados, reconhecendo aquele que deve obter o controle do barramento.

O dispositivo que obteve o controle do barramento passa a ser o iniciador. Ele entra na fase de *Seleção*, ativando o sinal SEL. Nessa fase, ele seleciona o alvo ativando as linhas de dados correspondentes ao seu próprio ID e ao identificador do alvo. Depois de certo tempo, ele desativa o sinal BSY. Quando o alvo detecta que a linha SEL está ativada, as linhas BSY e I/O estão desativadas, e reconhece seu ID, ele ativa o sinal BSY. Quando o iniciador detecta o sinal BSY, ele libera o barramento de dados e desativa na linha SEL.

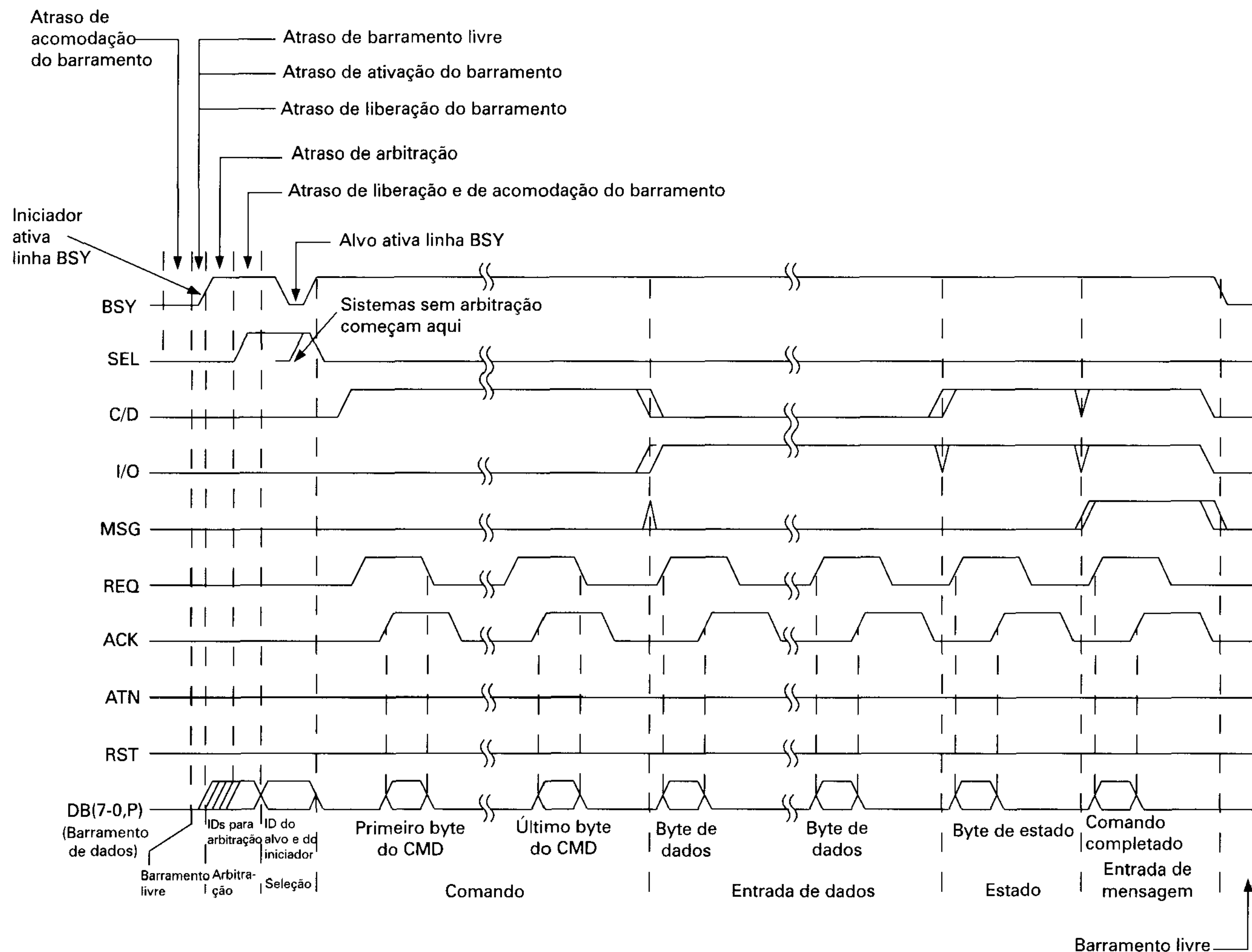


Figura 6.18 Exemplo de diagrama de tempo de uma operação de leitura no barramento SCSI.

Em seguida, o alvo entra na fase de *Comando*, ativando a linha C/D; essa linha permanece ativada durante toda essa fase. Ele então ativa a linha REQ para requisitar o primeiro byte do comando do iniciador. O iniciador coloca o primeiro byte do comando no barramento de dados e ativa a linha ACK. Depois de ler esse byte, o alvo desativa a linha REQ e o iniciador, a linha ACK. O primeiro byte do comando contém o código de operação do comando a ser executado e indica quantos bytes restam para serem transferidos. Os bytes restantes são transferidos seguindo o mesmo protocolo de comunicação, com as linhas REQ/ACK.

Depois de receber e interpretar o comando, o alvo coloca o barramento na fase de *Entrada de Dados*, desativando a linha C/D (o que indica que o barramento contém dados) e ativando a linha de I/O (o que indica que a transferência é do alvo para o iniciador). Ele coloca no barramento de dados o primeiro byte requisitado e ativa a linha REQ. Depois de ler o byte, o iniciador ativa a linha ACK. Os demais bytes de dados são transferidos seguindo o mesmo protocolo de sincronização com as linhas REQ/ACK.

Depois de transferir todos os dados requisitados, o alvo coloca o barramento na fase de *Estado* e envia um byte de estado para o iniciador, indicando que completou a transferência com sucesso. Nesse caso, a linha C/D é novamente ativada e a linha de I/O permanece ativada. O iniciador e o alvo usam as linhas REQ/ACK para coordenar a transferência dos bytes de estado.

Finalmente, o alvo coloca o barramento na fase de *Entrada de Mensagem*, ativando a linha MSG e transferindo um byte contendo a mensagem de *Comando Completado*. Quando esse byte é recebido pelo iniciador, o alvo desativa todos os sinais no barramento, restabelecendo a fase de *Barramento Livre*.

A transação representada na Figura 6.18 é um exemplo de transferência assíncrona. Uma transferência assíncrona requer o envio de sinais do protocolo de comunicação nas linhas REQ/ACK para cada byte transferido para sincronizar a comunicação. A interface SCSI também possibilita uma transferência síncrona, que requer menor sobrecarga. O modo de transferência síncrona é usado apenas para as fases de *Entrada de Dados* e de *Saída de Dados*. Como o modo padrão de transferência é assíncrono, a mudança para o modo síncrono deve ser negociada. Para isso, o alvo envia para o iniciador uma mensagem de *Requisição de Transferência de Dados Síncrona*, indicando seu tempo mínimo de transferência e seu tempo máximo de espera entre o envio de um sinal na linha REQ e o recebimento de um sinal na linha ACK correspondente. O iniciador responde com uma mensagem do mesmo tipo, indicando seu próprio tempo mínimo de transferência e seu tempo máximo de espera entre um sinal em REQ e um sinal em ACK.

Uma vez trocadas essas informações, a transferência passa a ser feita de modo síncrono, usando o maior dentre os dois tempos mínimos de transferência e o menor dos dois tempos de espera entre sinais nas linhas REQ e ACK. A transferência de dados procede da seguinte maneira. Os bytes são enviados, intercalados por um período de tempo igual ao tempo mínimo de transferência, pulsando a linha REQ a cada byte transferido. O receptor não precisa sinalizar imediatamente cada byte recebido, mas deve fazê-lo dentro do tempo máximo estabelecido entre sinais nas linhas REQ e ACK. O remetente pode, portanto, enviar um fluxo contínuo de bytes, desde que receba um sinal na ACK correspondente dentro do período de tempo estabelecido. Caso não seja recebido um sinal em ACK durante esse tempo, ele deve fazer uma pausa, iniciando nova sincronização por meio de sinais nas linhas REQ e ACK.

Mensagens

As mensagens trocadas entre o iniciador e o alvo têm como propósito gerenciar a interface SCSI. Existem três formatos de mensagens: de um byte, de dois bytes e mensagens estendidas de três ou mais bytes. Alguns exemplos de mensagens são:

- **Comando completado:** enviada pelo alvo para o iniciador para indicar que a execução do comando foi completada e que foi enviada uma informação de estado válido para o iniciador.
- **Desconexão:** enviada pelo alvo para informar ao iniciador que a conexão atual será interrompida, mas um restabelecimento da conexão será requisitado, mais tarde, para completar a operação corrente.
- **Detecção de erro pelo iniciador:** enviada pelo iniciador para informar ao alvo que ocorreu um erro (por exemplo, paridade), o que não impede que o alvo tente novamente a operação.
- **Aborto:** enviada do iniciador para o alvo para cancelar a operação corrente.
- **Transferência de dados síncrona:** trocada entre o iniciador e o alvo para estabelecer o modo de transferência de dados síncrona.

Comandos

O núcleo do protocolo SCSI é seu conjunto de comandos. Um comando é enviado por um iniciador para requisitar alguma ação de um alvo. O comando pode envolver a obtenção de dados do alvo (leitura), o envio de dados para o alvo (escrita) ou alguma outra ação específica de um periférico particular. Em todos os casos, a execução de qualquer comando envolve alguns ou todos os seguintes passos:

- O alvo recebe e decodifica o comando.
- Dados são transferidos de e para o alvo (não ocorre em todos os comandos).
- O alvo gera informação de estado e retorna essa informação para o iniciador.

Um comando é enviado na forma de um bloco descritor de comando (BDC — *command descriptor block*), preparado pelo iniciador. Uma vez estabelecida a conexão entre um iniciador e um alvo, para iniciar a execução de um comando, o iniciador transfere um BDC para o alvo por meio do barramento de dados.

A Figura 6.19 mostra o formato geral do BDC, que é constituído dos seguintes campos:

- **Código de operação:** especifica um comando particular. O código de operação determina também o restante do BDC.
- **Número de unidade lógica:** identifica um dispositivo, físico ou virtual, conectado a um alvo. Os números de unidade lógica podem ser usados para endereçar múltiplos dispositivos que compartilhem um mesmo controlador ou para endereçar múltiplos volumes lógicos em um disco.
- **Endereço de bloco lógico:** no caso de operações de leitura ou de escrita, o BDC inclui o endereço lógico inicial da transferência de dados.
- **Tamanho total da transferência:** em operações de leitura ou de escrita, esse campo especifica o número de blocos lógicos contíguos de dados a serem transferidos.

- **Tamanho da lista de parâmetros:** especifica o número de bytes enviados durante a fase de *Saída de Dados*. Esse campo é usado, tipicamente, para os parâmetros enviados para um alvo (por exemplo, parâmetros de modo, diagnóstico ou relatório).
- **Tamanho da área alocada:** especifica o número máximo de bytes alocados pelo iniciador para os dados retornados.
- **Controle:** esse campo inclui os bits LINK e FLAG, que controlam o mecanismo de ligação de comandos. Se o bit LINK estiver ativado, o comando corrente não termina com a fase de *Barramento Livre*, mas inicia imediatamente a execução do comando seguinte, iniciando uma nova fase de *Comando*. Se o bit FLAG estiver ativado, o alvo envia a mensagem de *Comando Ligado Completado*, caso o comando seja completado com sucesso, e retorna o mesmo valor do bit FLAG do BDC. Tipicamente, o valor 1 no bit FLAG é usado pelo iniciador para causar uma interrupção no iniciador entre dois comandos ligados.

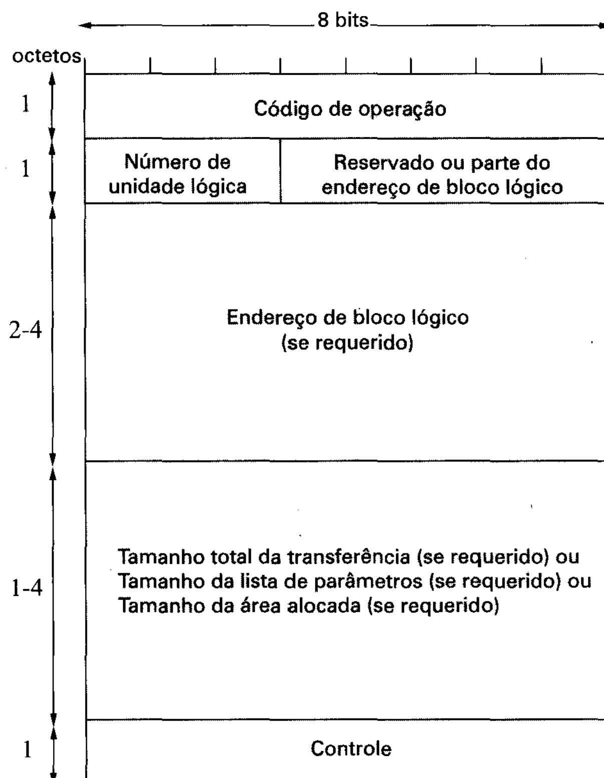


Figura 6.19 Formato de bloco descritor de comando da SCSI.

A especificação SCSI define uma ampla variedade de tipos de comandos. A maioria é específica de um tipo particular de dispositivo. O padrão SCSI-2 inclui comandos para os seguintes tipos de dispositivos:

- Dispositivos de acesso direto
- Dispositivos de acesso seqüencial

- Impressoras
- Processadores
- Dispositivos de escrita única
- CD-ROMs
- Scanners
- Dispositivos de memória óptica
- Dispositivos com troca de mídia
- Dispositivos de comunicação

Além disso, existe um conjunto de 17 comandos que se aplicam a qualquer tipo de dispositivo. Dentre esses, quatro são obrigatórios e devem ser implementados por todos os dispositivos:

- **Interrogação:** requisita que os parâmetros do alvo e dos dispositivos periféricos a ele conectados sejam enviados para o iniciador.
- **Requisição de estado:** requisita ao alvo que transfira dados de estado para o iniciador. Dados de estado incluem condições de erros (por exemplo, falta de papel), informações de posicionamento (por exemplo, fim de uma mídia) e informação de estado lógico (por exemplo, o comando corrente atingiu uma marca de arquivo).
- **Envio de diagnóstico:** requisita ao alvo que efetue testes de diagnóstico sobre si próprio, sobre os dispositivos periféricos a ele conectados ou sobre ambos.
- **Teste de unidade pronta:** possibilita verificar se uma unidade lógica está pronta.

O repertório de comandos SCSI é a parte mais importante da especificação. Ele apresenta uma forma padronizada para tratar os dispositivos periféricos mais comuns de computadores pessoais e estações de trabalho, simplificando a tarefa de implementar software de E/S no sistema hospedeiro.

Barramento serial *FireWire*

Com as velocidades dos processadores atingindo 100 MHz* e os dispositivos de armazenamento com capacidade de vários gigabits, as demandas por E/S nos PCs, nas estações de trabalho e nos servidores são formidáveis. As tecnologias de canal de E/S de alta velocidade desenvolvidas para sistemas de grande porte e supercomputadores são ainda muito caras e volumosas para serem usadas nesses sistemas menores. Por isso, há grande interesse em desenvolver uma alternativa para a SCSI e outras interfaces de E/S de sistemas de pequeno porte, que possibilite a transferência de dados em alta velocidade. Uma proposta é o padrão IEEE 1394, que especifica um barramento serial de alto desempenho, conhecido como *FireWire.*

O *FireWire* apresenta diversas vantagens sobre a interface SCSI e outras interfaces de E/S. Ele tem velocidade muito alta, baixo custo e é de fácil implementação. Tem sido usado não apenas em sistemas de computação mas também em produtos eletrônicos, tais como câmeras digitais, videocassetes e televisores. Nesses produtos, o *FireWire* é usado para transferir imagens de vídeo, produzidas cada vez mais a partir de fontes digitalizadas.

Uma das vantagens da interface *FireWire* vem, principalmente, do fato de utilizar transmissão serial (um bit de cada vez), e não paralela. Interfaces paralelas, tais com a SCSI, requerem maior número de fios, o que leva a cabos mais caros e mais largos, além de conectores mais caros e com maior número de pinos, sujeitos a entortar ou quebrar. Um cabo com mais fios necessita também de uma blindagem para evitar interferências elétricas entre os fios. A

* N.R.T.: Atualmente, os processadores já atingiram velocidades de cerca de 2000 MHz ou 2 GHz.

interface paralela requer ainda a sincronização entre os fios, problema que se torna mais grave à medida que se utilizam de cabos mais longos.

Além disso, os computadores estão cada vez menores, apesar de terem maior poder computacional e maior demanda por E/S. Computadores portáteis pequenos, como os *handhelds* e os *palmtops*, têm pequeno espaço para conectores, mesmo necessitando de altas taxas de transferência de dados, para manipular imagens e vídeo.

A intenção do *FireWire* é oferecer uma interface de E/S única, com um conector simples que possa manipular grande número de dispositivos por meio de uma única porta, de modo que os conectores para componentes como o mouse, a impressora a laser, a SCSI, as unidades de discos externas, som e redes locais possam ser substituídos por esse único conector. O conector é inspirado em um modelo usado no Game Boy da Nintendo. Ele é tão conveniente que o usuário pode localizá-lo atrás da máquina e encaixá-lo mesmo sem olhar.

Configurações *FireWire*

O *FireWire* usa uma configuração para conexão de dispositivos na forma de cadeia circular, possibilitando a conexão de até 63 dispositivos a uma única porta. Além disso, até 1022 barramentos *FireWire* podem ser conectados entre si, usando pontes, o que possibilita ao sistema possuir tantos periféricos quantos forem necessários.

O *FireWire* fornece um tipo de conexão conhecido como *hot plugging* (conexão a quente), que possibilita conectar ou desconectar periféricos sem desligar ou reconfigurar o sistema de computação. Além disso, ele permite uma configuração automática, não sendo necessário especificar manualmente os IDs de cada dispositivo ou se preocupar com suas posições relativas. A Figura 6.20 faz uma comparação entre o *FireWire* e a SCSI. Na interface SCSI, as duas extremidades do barramento devem ter terminadores e a configuração especifica um endereço distinto para cada dispositivo. No *FireWire*, não existem terminadores e o sistema efetua a configuração automaticamente, designando endereços aos dispositivos conectados. Note também que um barramento *FireWire* não precisa ser configurado rigorosamente na forma de uma cadeia circular. É também possível usar uma configuração com uma estrutura de árvore.

Uma importante característica do padrão *FireWire* é especificar um conjunto de três camadas de protocolos, para padronizar o modo como o sistema hospedeiro interage com os dispositivos periféricos, através do barramento serial. A Figura 6.21 mostra essa pilha de protocolos. As três camadas dessa pilha são descritas a seguir:

- **Camada física:** define os meios de transmissão permitidos sob o *FireWire* e as características elétricas e de sinalização de cada um.
- **Camada de enlace:** descreve a transmissão de dados em pacotes.
- **Camada de transação:** define um protocolo do tipo requisição-resposta, que esconde os detalhes das camadas inferiores do *FireWire* das aplicações.

Camada física

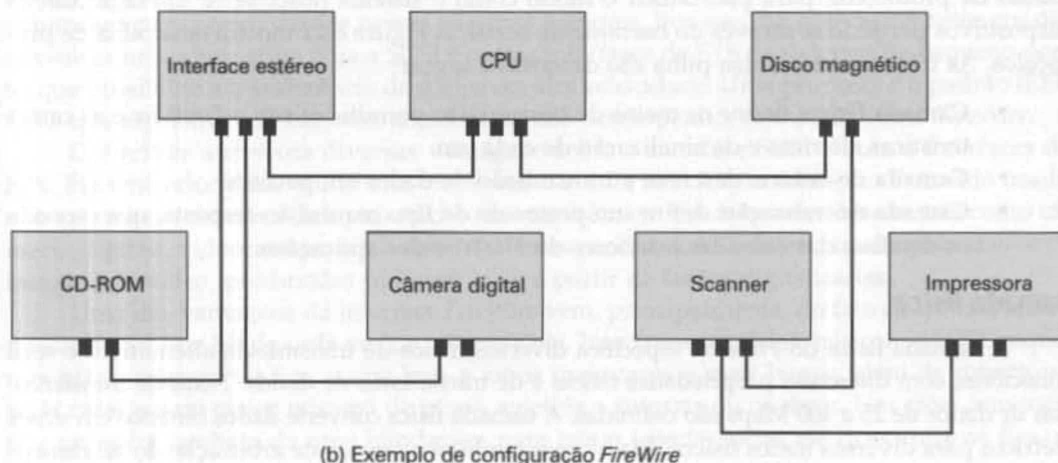
A camada física do *FireWire* especifica diversos meios de transmissão alternativos e seus conectores, com diferentes propriedades físicas e de transmissão de dados. Taxas de transferências de dados de 25 a 400 Mbps são definidas. A camada física converte dados binários em sinais elétricos para diversos meios físicos. Ela oferece também o serviço de arbitração do barramento, o que garante que apenas um dispositivo transmitirá os dados de cada vez.

O *FireWire* permite duas formas de arbitração. A mais simples é baseada no arranjo dos nós do barramento *FireWire* em estrutura de árvore, como mencionado anteriormente. A forma de cadeia circular é um caso especial dessa estrutura de árvore. A camada física contém circuitos lógicos que possibilitam que todos os dispositivos conectados configurem a si próprios, de modo que um deles seja designado como a raiz da árvore e os demais sejam organizados em uma relação pai/filho, formando a topologia de árvore. Uma vez estabelecida essa configuração, o nó raiz da árvore age como um árbitro central, processando as requisições de acesso ao barramento, em uma estratégia baseada na ordem de chegada (a primeira a chegar é a primeira a ser atendida). No caso de requisições simultâneas, o acesso é cedido ao nó de prioridade mais alta. As prioridades são estabelecidas atribuindo as mais altas aos nós que estão mais próximos da raiz e, entre os nós localizados à mesma distância da raiz, o de prioridade mais alta é aquele cujo número de ID é menor.

Esse método de arbitração é suplementado por duas funções adicionais: arbitração imparcial e arbitração urgente. Na arbitração imparcial, o tempo do barramento é dividido em *intervalos imparciais*. No início de um intervalo, cada nó ativa um sinal de habilitação de arbitração. Durante esse intervalo, cada nó pode competir pelo acesso ao barramento. Uma vez que um nó obtenha acesso ao barramento, ele desativa seu sinal de habilitação de barramento e não pode competir novamente por um acesso imparcial nesse intervalo. Esse esquema torna a arbitração mais imparcial, evitando que um ou mais dispositivos de alta prioridade monopolizem o barramento.



(a) Exemplo de configuração SCSI



(b) Exemplo de configuração *FireWire*

Figura 6.20 Comparação entre configurações SCSI e *FireWire*.

Além do esquema de arbitração imparcial, alguns dispositivos podem ser configurados com prioridade *urgente*. Esses nós podem obter várias vezes o controle do barramento durante um intervalo imparcial. Em essência, é usado um contador em cada nó de alta prioridade, que possibilita a esses nós de alta prioridade controlar 75% do tempo disponível do barramento. Para cada pacote transmitido como não-urgente, três podem ser transmitidos como urgentes.

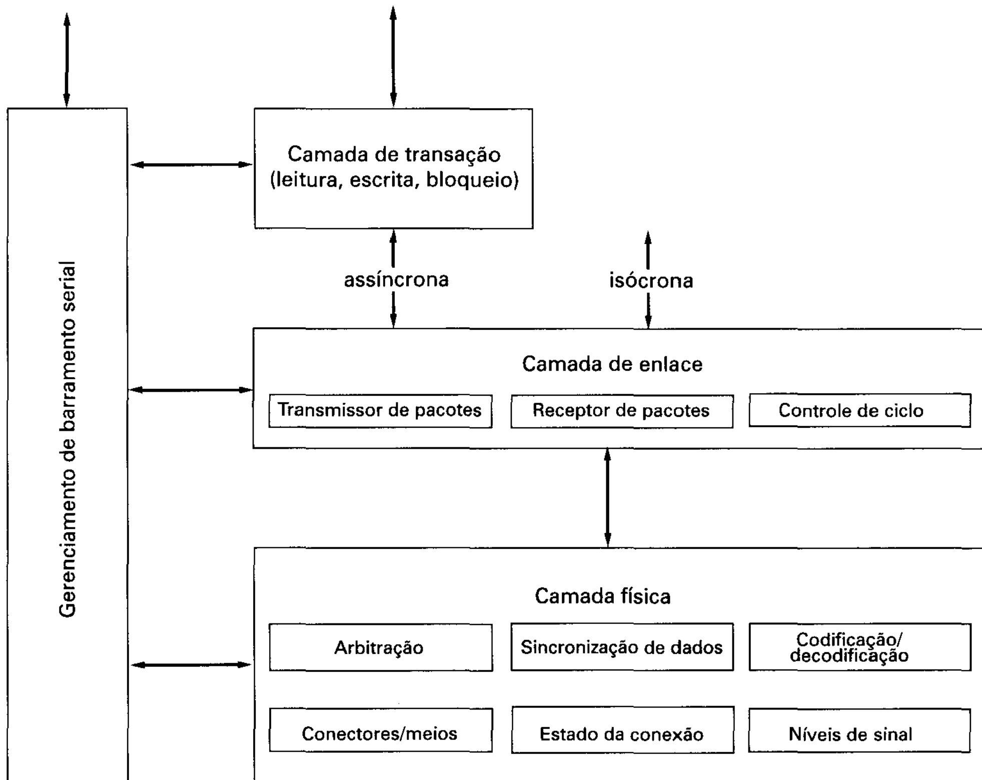


Figura 6.21 Pilha de protocolos *FireWire*.

Camada de enlace

A camada de enlace define a transmissão de dados na forma de pacotes. Ela possibilita dois tipos de transmissão:

- **Assíncrona:** uma quantidade variável de dados e diversos bytes de informação da camada de transação são transferidos, sob a forma de um pacote, para um endereço específico, e um pacote de reconhecimento (ACK) é retornado.
- **Isócrona:** uma quantidade variável de dados é transferida em uma seqüência de pacotes de tamanho fixo, transmitidos a intervalos regulares. Esse meio de transmissão usa um endereçamento simplificado e não necessita de reconhecimento.

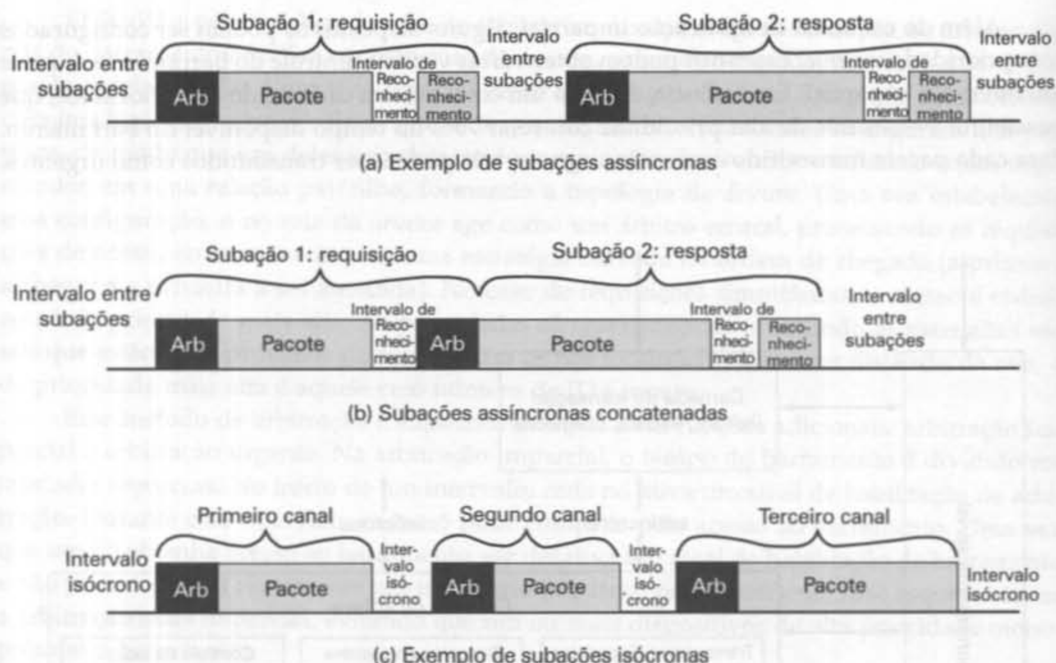


Figura 6.22 Subações no *FireWire*.

A transmissão assíncrona é usada para dados que necessitam de uma taxa de transferência fixa. Tanto o esquema de arbitração imparcial quanto o de arbitração urgente podem ser usados para esse tipo de transmissão. O método-padrão é a arbitração imparcial. Um dispositivo que necessita usar uma fração substancial do tempo do barramento ou que tenha restrições severas de tempo sobre a latência deve adotar o método de arbitração urgente. Por exemplo, um nó de alta velocidade, destinado à coleta de dados de tempo real, pode usar a arbitração urgente, quando suas áreas de armazenamento temporário de dados críticos estiverem mais da metade ocupadas.

A Figura 6.22a apresenta uma transação assíncrona típica. O processo de entrega de um único pacote é conhecido como subação. Uma subação é constituída de cinco períodos de tempo:

- **Seqüência de arbitração:** é a troca de sinais necessária para determinar o dispositivo que obterá o controle do barramento.
- **Transmissão de pacote:** cada pacote inclui um cabeçalho, que contém os IDs dos nós fonte e destino. O cabeçalho também contém informação sobre o tipo do pacote, um código CRC de verificação de erros (verificação por redundância cíclica) e parâmetros específicos do tipo de pacotes. Um pacote pode também incluir um bloco de dados, constituído de dados do usuário e de outro CRC.
- **Intervalo de reconhecimento:** é o tempo de retardo para que o destino receba e decodifique um pacote e envie outro de reconhecimento (ACK).
- **Reconhecimento:** o dispositivo que recebeu o pacote retorna outro de reconhecimento com um código que indica a ação que foi tomada.

- **Intervalo entre subações:** é um período ocioso obrigatório para assegurar que outros nós conectados ao barramento não iniciem uma arbitração, antes de o pacote de reconhecimento ter sido transmitido.

No instante em que o pacote de reconhecimento é enviado, o nó emitente do pacote detém o controle do barramento. Por isso, se a troca de pacotes é uma interação de requisição/resposta entre os dois nós, o respondedor pode imediatamente transmitir o pacote de resposta sem que haja uma seqüência de arbitração (Figura 6.22.b).

Para dispositivos que geram ou consomem dados regularmente, tais como dispositivos de som ou vídeo digital, é usado o acesso isócrono. Esse método garante que os dados sejam transmitidos dentro de um intervalo de latência especificado, com uma taxa garantida de transferência de dados.

Para acomodar uma carga de tráfego composta de fontes de dados assíncronas e isócronas, um dos nós é designado como *mestre de ciclo.* Periodicamente, o mestre de ciclo envia um pacote de início de ciclo, que sinaliza o início de um ciclo isócrono para os demais nós. Durante esse ciclo, apenas pacotes isócronos podem ser enviados (Figura 6.22c). Cada fonte de dados isócrona compete pelo controle do barramento. O nó que obtém o acesso transmite imediatamente um pacote. Como não é requerido reconhecimento desse pacote, logo depois que um pacote isócrono é transmitido, outras fontes de dados isócronas competem pelo controle do barramento. Como resultado, existe um pequeno intervalo entre a transmissão de um pacote e o período de arbitração para envio do próximo pacote, ditado por atrasos no barramento. Esse atraso, conhecido como intervalo isócrono, é menor que o atraso entre subações.

Depois que todas as fontes isócronas são transmitidas, o barramento permanece ocioso tempo suficiente para que aconteça um intervalo entre subações. Esse é o sinal para que as fontes assíncronas possam agora competir pelo controle do barramento. As fontes assíncronas podem então usar o barramento até o início do próximo ciclo isócrono.

Pacotes isócronos são rotulados com números de canal de 8 bits, previamente estabelecidos por meio de um diálogo entre os dois nós que estão para iniciar a troca de dados isócrona. O cabeçalho, que é menor que o de um pacote assíncrono, inclui também um campo de tamanho dos dados e um CRC do cabeçalho.

6.8 LEITURA E SITES WEB RECOMENDADOS

Uma boa discussão sobre a arquitetura e os módulos de E/S dos processadores Intel, incluindo o 82C59A e o 82C55A, pode ser encontrada em Brey (1997).

Dois livros que contêm uma boa introdução à interface SCSI são de autoria de Schmidt (1997) e NCR Corporation (1990). O *FireWire* é abordado detalhadamente em Anderson (1998b).



Sites Web recomendados:

- **T10 Home Page:** T10 é um comitê técnico do Comitê Nacional de Padrões em Tecnologia da Informação (National Committee on Information Technology Standards), responsável por interfaces de baixo nível. Seu principal trabalho é a interface SCSI.
- **SCSI Trade Association:** inclui informação técnica e indicações de vendedores.
- **1394 Trade Association:** inclui informação técnica e indicações de vendedores do *FireWire.*

6.9 EXERCÍCIOS

6.1 A Seção 6.3 relacionou uma vantagem e uma desvantagem da E/S mapeada na memória em relação à E/S independente. Enumere mais duas vantagens e duas desvantagens.

6.2 Em quase todos os sistemas que incluem módulos de DMA, o acesso do módulo de DMA à memória principal tem prioridade mais alta do que o acesso da CPU. Por quê?

6.3 Considere o sistema de disco descrito no Exercício 5.6 e suponha que o disco gira a 360 rpm. Um processador lê um setor do disco usando E/S dirigida por interrupção, com uma interrupção por byte transferido. Se o processador gasta 2,5 μs para processar cada interrupção, qual a porcentagem do tempo do processador despendida no tratamento de E/S (desconsidere o tempo de busca no disco)?

6.4 Repita o Exercício 6.3 usando DMA e supondo uma interrupção a cada setor transferido.

6.5 Um módulo de DMA transfere caracteres para a memória usando a técnica de roubo de ciclo, a partir de um dispositivo que transfere dados à taxa de 9600 bps. O processador busca instruções a uma taxa de 1 milhão de instruções por segundo (1 MIPS). Qual é a diminuição na velocidade do processador em virtude da atividade do módulo de DMA?

6.6 Um computador de 32 bits possui dois canais seletores e um canal multiplexador. Cada canal seletor contém dois discos magnéticos e duas unidades de fita magnética. Ao canal multiplexador são conectadas duas impressoras de linha, duas leitoras de cartão e dez terminais de vídeo/teclado. Considere as seguintes taxas de transferência:

Unidade de disco:	800 Kbytes/s
Unidade de fita magnética:	200 Kbytes/s
Impressora de linha:	6,6 Kbytes/s
Leitora de cartão:	1,2 Kbyte/s
Terminal de vídeo/teclado:	1 Kbyte/s

Estime a taxa máxima agregada de transferência de E/S nesse sistema.

6.7 Um computador contém um processador e um dispositivo D de E/S, conectados à memória principal M por meio de um barramento compartilhado com uma largura do barramento de dados de um palavra. O processador pode executar no máximo 10^6 instruções por segundo. Uma instrução requer, em média, cinco ciclos de máquina, três dos quais usam o barramento de memória. Uma operação de leitura ou de escrita na memória gasta um ciclo de máquina. Suponha que o processador execute programas continuamente, que consomem 95% da sua taxa de execução de instruções, mas não envolvem qualquer instrução de E/S. Suponha, ainda, que o ciclo do processador tem a mesma duração do ciclo de barramento e que o dispositivo de E/S é para ser usado para transferir grandes blocos de dados entre M e D.

a. Supondo que é usada E/S programada e que a transferência de E/S de uma palavra requer a execução de duas instruções pelo processador, estime a taxa máxima de transferência de dados de E/S em palavras por segundo através de D.

b. Estime essa mesma taxa supondo que é usado acesso direto à memória.

6.8 Uma fonte de dados produz caracteres ASCII de 7 bits, a cada qual é anexado um bit de paridade. Obtenha uma expressão para a taxa efetiva máxima de transferência de dados (taxa de transferência de bits de caracteres ASCII), sobre uma linha de R bps, para os seguintes casos:

a. Transmissão assíncrona, com um bit de parada a cada 1,5 unidade de dados.

b. Transmissão síncrona de bits, com um quadro de transmissão de 48 bits de controle e 128 bits de informação.

c. O mesmo que em (b), mas com um campo de informação de 1024 bits.

d. Transmissão síncrona de caracteres, com um quadro de transmissão de 9 caracteres de controle e 16 caracteres de informação.

e. O mesmo que em (d), com campo de informação de 128 caracteres.

6.9 O problema a seguir é baseado em uma metáfora dos mecanismos de E/S sugerida em Erkert (1990) (Figura 6.23):

Dois garotos estão jogando, um de cada lado de uma cerca alta. Um deles, chamado Servidor de maçãs, tem um belo pé de maçãs, carregado com deliciosas maçãs, do seu lado da cerca; ele sente-se feliz em fornecer maçãs ao outro garoto sempre que ele solicita. O outro garoto, chamado Comedor de maçãs, adora comer maçãs, mas não tem nenhuma. De fato, ele deve comer maçãs a uma taxa fixa (uma maçã por dia mantém o médico longe). Se comer maçãs a uma taxa mais alta, ele ficará doente. Se comer mais devagar, sofrerá de desnutrição. Nenhum dos garotos pode falar e, portanto, o problema é transferir maçãs do Servidor para o Comedor de maçãs na taxa correta.

a. Suponha que existe um relógio com alarme no alto da cerca, que pode ser programado para disparar o alarme. Como o relógio pode ser usado para resolver o problema? Desenhe um diagrama de tempo para ilustrar a solução.

b. Suponha agora que não existe nenhum relógio. Em vez disso, o Comedor de maçãs tem uma bandeira, que ele pode balançar quando desejar uma maçã. Sugira uma nova solução. Seria útil que o Servidor de maçãs também tivesse uma bandeira? Em caso afirmativo, incorpore isso à sua solução. Discuta as desvantagens dessa abordagem.

c. Agora dispense a bandeira e suponha que exista um longo pedaço de corda. Sugira uma solução que utilize a corda e seja melhor do que a apontada em (b).

6.10 Suponha que um microprocessador de 16 bits e dois de 8 bits devam ser conectados a um barramento do sistema. Considere os seguintes detalhes:

1. Todos os microprocessadores possuem as características de hardware necessárias para qualquer tipo de transferência de dados: E/S programada, E/S dirigida por interrupção e DMA.

2. Todos os microprocessadores têm barramento de endereço de 16 bits.

3. Duas placas de memória, cada uma com capacidade de 64 Kbytes, são conectadas ao barramento. O projetista quer usar uma memória compartilhada que seja a maior possível.

4. O barramento do sistema contém, no máximo, quatro linhas de interrupção e uma linha de DMA. Faça quaisquer outras hipóteses que julgar necessárias.

a. Especifique o tipo e o número de linhas do barramento do sistema.

b. Explique como os dispositivos relacionados acima são conectados ao barramento do sistema.

Fonte: Alexandridis (1993).

6.11 Em um barramento SCSI, cada dispositivo de E/S negocia com o computador hospedeiro a taxa de transferência de dados em lotes a serem usados entre eles – em geral a maior taxa suportada por ambos. Suponha que a taxa máxima de transferência de dados em lotes do computador hospedeiro seja de 20 Mbytes/s. Suponha também que todos os dispositivos de E/S possuam áreas internas de armazenamento temporário de dados, de tamanho adequado para que possam manter sua taxa de transferência sustentada de dados, mesmo quando estão competindo por acesso ao barramento.

a. Suponha que uma unidade de fita esteja conectada ao SCSI, com uma taxa de transferência sustentada de 500 Kbytes/s e taxa de transferência em lotes de 4 Mbytes/s. Você deseja acrescentar, ao mesmo barramento, algumas unidades de disco, cada uma com uma taxa de transferência sustentada de 6 Mbytes/s e uma taxa de transferência em lotes de 20 Mbytes/s. Quantas unidades de disco podem ser conectadas ao barramento, de modo que todos os dispositivos ainda possam operar simultaneamente em velocidade total? Qual é a utilização do barramento nesse caso?
Sugestão: determine a porcentagem de tempo que cada dispositivo requer para transferir todos os seus dados. (Nota: use 1 K = 1000, em vez de 1024, e 1 M = 1.000.000, em vez de 1.048.576; essa aproximação é suficiente para o propósito desse problema. Taxas de transferências são normalmente expressas na base decimal, enquanto capacidades de áreas de armazenamento temporário e de tamanhos de transferências de dados são em geral expressas na base binária.)

b. Agora refina o modelo ligeiramente, supondo que a unidade de fita requeira uma sobrecarga de 4 ms para cada transferência e que cada transferência tenha tamanho máximo de 64 Kbytes. As unidades de disco também necessitam de uma sobrecarga de 4 ms por transferência, mas podem transferir até 256 Kbytes por requisição. Reavalie a utilização do barramento para cada dispositivo e a utilização total. O barramento ainda é adequado para o número de dispositivos especificados na sua resposta para o item (a)?

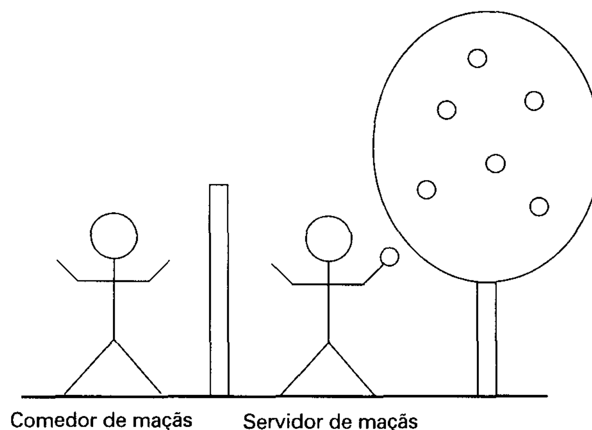
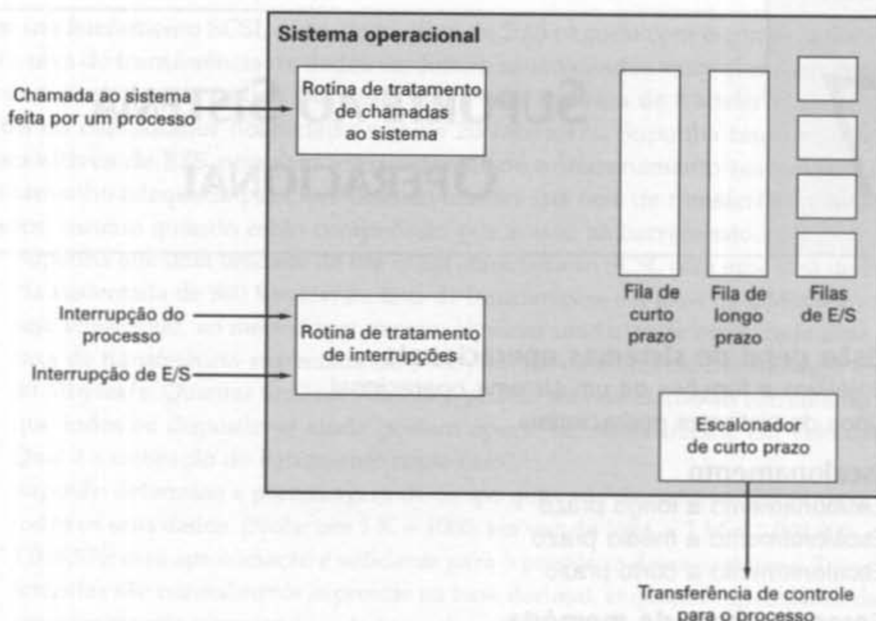


Figura 6.23 Problema da maçã.

capítulo

7

SUPORTE AO SISTEMA OPERACIONAL



- O sistema operacional (SO) é o software que controla a execução de programas em um processador e gerencia os recursos do computador. Diversas funções desempenhadas pelo SO, incluindo o escalonamento de processos e o gerenciamento de memória, só podem ser executadas de modo rápido e eficiente se o SO dispuser de um suporte adequado do hardware do processador. Quase todos os processadores dispõem desse suporte, em maior ou menor extensão, incluindo hardware de gerenciamento de memória virtual e de gerenciamento de processos. Isso inclui registradores de propósito especial e áreas de armazenamento temporário, além de um conjunto de circuitos para realizar tarefas básicas de gerenciamento de recursos.
- Uma das funções mais importantes de um SO é o escalonamento de processos ou tarefas. O SO determina quais processos devem ser executados a cada instante. Tipicamente, o hardware interrompe periodicamente o processo que está sendo executado para permitir ao SO realizar uma nova decisão de escalonamento. Isso possibilita compartilhar o tempo do processador entre um determinado número de processos de modo imparcial.
- Outra função importante de um sistema operacional é o gerenciamento de memória. A maioria dos sistemas operacionais atuais inclui a capacidade de memória virtual, o que traz dois benefícios: (1) um processo pode ser executado na memória principal sem que todas as instruções e dados do programa precisem estar armazenados na memória principal; e (2) o espaço de memória total disponível para um programa pode exceder o tamanho da memória principal do sistema. Embora o gerenciamento de memória seja feito por software, o sistema operacional conta com suporte do hardware do processador, incluindo hardware de paginação e de segmentação da memória.

Embora o foco deste livro seja o hardware do computador, existe outra área que precisa também ser abordada: o sistema operacional. O sistema operacional é um programa que gerencia os recursos do computador, fornece serviços para os programadores e estabelece uma ordem de execução de outros programas. É essencial certo conhecimento sobre sistemas operacionais, para que se possa entender os mecanismos pelos quais a CPU controla o computador. Em particular, o efeito das interrupções e o gerenciamento da hierarquia de memória são mais bem explicados nesse contexto.

Este capítulo apresenta, inicialmente, uma visão geral e um breve histórico de sistemas operacionais. A maior parte do capítulo é voltada para as duas funções do sistema operacional que são mais relevantes para o estudo da arquitetura e organização de computadores: o escalonamento de tarefas e o gerenciamento de memória.

7.1 VISÃO GERAL DE SISTEMAS OPERACIONAIS

Objetivos e funções de um sistema operacional

Um sistema operacional é um programa que controla a execução de programas aplicativos e age como uma interface entre o usuário e o hardware do computador. Ele possui dois objetivos:

- **Conveniência:** um sistema operacional visa tornar o uso do computador mais conveniente.
- **Eficiência:** um sistema operacional permite uma utilização mais eficiente dos recursos do sistema.

Esses dois aspectos são examinados separadamente a seguir.

O sistema operacional como uma interface entre usuário e computador

O hardware e o software usados para fornecer aplicações aos usuários podem ser vistos sob a forma de uma organização em camadas ou hierárquica, como representado na Figura 7.1. O usuário dessas aplicações, o usuário final, geralmente não está interessado na arquitetura do computador. Dessa maneira, ele vê o sistema de computação em termos de uma aplicação. Essa aplicação é escrita em uma linguagem de programação e desenvolvida por um programador de aplicações. Se esses programas aplicativos tivessem de ser escritos usando o conjunto de instruções do processador e, além disso, tivessem também de controlar o hardware do computador, a tarefa de desenvolver programas seria extremamente complexa. Para facilitar essa tarefa, existe um conjunto de programas de sistema. Alguns desses programas são conhecidos como utilitários. Eles implementam funções usadas freqüentemente que auxiliam na criação de programas, gerenciamento de arquivos e controle de dispositivos de E/S. Um programador usa esses recursos para desenvolver uma aplicação, que, ao ser executada, invoca os utilitários para desempenhar certas funções. O programa de sistema mais importante é o sistema operacional, o qual esconde os detalhes do hardware do programador, fornecendo uma interface conveniente para o uso do sistema. Ele age como um mediador, tornando mais fácil para o programador e para os programas aplicativos acessar e usar esses recursos e serviços.

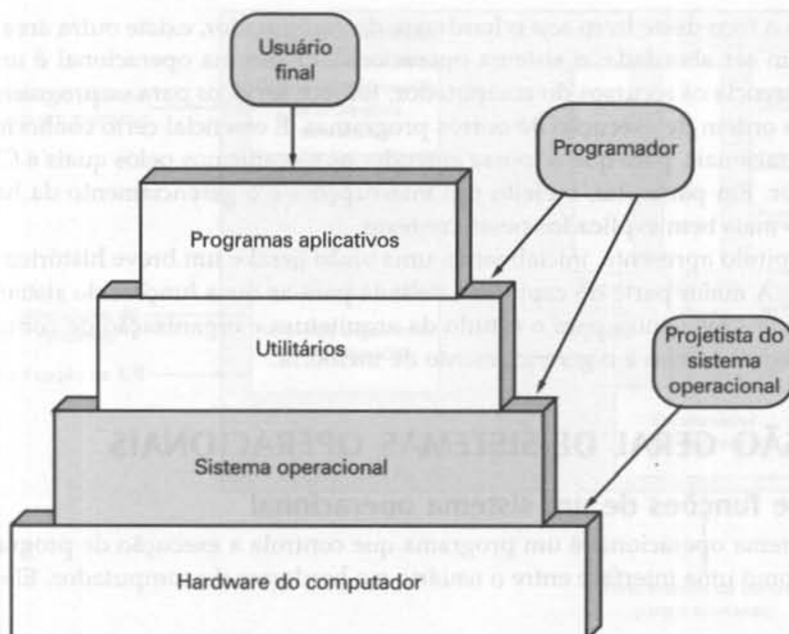


Figura 7.1 Camadas e visões de um sistema de computação.

O sistema operacional tipicamente fornece serviços para a realização das seguintes atividades:

- **Criação de programas:** o sistema operacional apresenta uma variedade de recursos e serviços para auxiliar o programador no desenvolvimento de programas, tais como editores e depuradores. Esses serviços tipicamente são oferecidos na forma de programas utilitários, que na verdade não são parte do sistema operacional, mas podem ser acessados por meio dele.
- **Execução de programas:** várias tarefas precisam ser realizadas para que um programa possa ser executado. Instruções e dados devem ser carregados na memória principal, dispositivos de E/S e arquivos precisam ser inicializados e outros recursos devem ser preparados. O sistema operacional realiza todas essas tarefas para o usuário.
- **Acesso a dispositivos de E/S:** cada dispositivo de E/S possui seu próprio conjunto peculiar de instruções ou sinais de controle para operação. O sistema operacional cuida dos detalhes do uso de cada dispositivo, de modo que o programador possa pensar apenas em termos de operações simples de leitura e de escrita.
- **Acesso controlado aos arquivos:** no caso de arquivos, o controle deve incluir não apenas um entendimento sobre a natureza do dispositivo de E/S (unidade de disco ou de fita) mas também sobre o formato dos arquivos no meio de armazenamento. Mais uma vez, o sistema operacional cuida dos detalhes. Além disso, no caso de sistemas usados simultaneamente por vários usuários, ele fornece mecanismos de proteção para o controle de acesso aos arquivos.
- **Acesso ao sistema:** no caso de sistemas compartilhados ou públicos, o sistema operacional controla o acesso ao sistema como um todo e o acesso a recursos específicos.

A função de acesso deve fornecer proteção contra o uso não-autorizado tanto para recursos quanto para dados de usuários e resolver conflitos em caso de contenção de um recurso.

- **Detecção e reação aos erros:** diversos erros podem ocorrer durante a operação de um sistema de computação, incluindo erros de hardware internos e externos, tais como erro de memória e falha ou mau funcionamento de dispositivo, assim como vários erros de software, tais como *overflow* em operação aritmética, tentativa de endereçar uma área de memória não permitida e a impossibilidade de o sistema operacional atender a uma requisição de uma aplicação. Em cada caso, o sistema operacional deve reagir no sentido de eliminar a condição de erro, com o menor impacto possível sobre as aplicações em execução. Essa reação pode variar desde terminar a execução do programa que causou o erro até tentar executar novamente a operação ou, simplesmente, relatar a ocorrência do erro à aplicação.
- **Monitoração:** um bom sistema operacional mantém estatísticas de uso de vários recursos e monitora parâmetros de desempenho, tais como o tempo de resposta. Em qualquer sistema, essa informação é útil para antecipar a necessidade de futuros melhoramentos e para a sintonia do sistema para aumentar seu desempenho. Em um sistema multiusuário, essa informação pode também ser usada para tarifação pela utilização de recursos.

O sistema operacional como gerente de recursos

Um computador é um conjunto de recursos, usados para processar, transferir e armazenar dados, assim como para controlar essas funções. O sistema operacional é responsável por gerenciar o uso desses recursos.

Podemos realmente dizer que é o sistema operacional que controla o processamento, o armazenamento e a transferência de dados? De certo ponto de vista, a resposta é sim: gerenciando os recursos do computador, o sistema operacional detém o controle das funções básicas desse computador. Mas esse controle é exercido de uma maneira curiosa. Normalmente, pensamos no mecanismo de controle como algo externo ao que é controlado ou, pelo menos, como algo que é uma parte distinta e separada do que é controlado (por exemplo, um sistema de aquecimento residencial é controlado por um termostato, que é completamente distinto do sistema de geração de calor e do aparato de distribuição de calor). Esse não é o caso do sistema operacional, que, como mecanismo de controle, é incomum em dois aspectos:

- O sistema operacional é um programa como outro qualquer, sendo executado pelo processador.
- O sistema operacional freqüentemente renuncia ao controle do processador para, em seguida, obter o controle novamente.

O sistema operacional é, de fato, nada mais que um programa de computador. Assim como outros programas, ele contém instruções para o processador. A diferença-chave está na intenção do programa. O sistema operacional direciona o processador no uso dos recursos do sistema, assim como na execução de outros programas. Mas, para que o processador possa executar outros programas, ele deve interromper a execução do sistema operacional. Dessa maneira, o sistema operacional libera o controle ao processador, para que ele possa executar algum trabalho 'útil', e então retoma o controle por um tempo suficiente para preparar o pro-

cessador para executar uma próxima tarefa. O mecanismo usado para isso deve tornar-se mais claro ao longo deste capítulo.

A Figura 7.2 mostra os principais recursos gerenciados pelo sistema operacional. Uma parte do sistema operacional reside na memória principal. Essa parte inclui o **núcleo** (*kernel*), que contém as funções do sistema operacional usadas mais freqüentemente, além de outras partes do sistema operacional que estão em uso naquele momento. O restante da memória principal contém outros dados e programas de usuário. Como veremos mais adiante, a alocação desse recurso (a memória principal) é controlada, em conjunto, pelo sistema operacional e pelo hardware de gerenciamento de memória do processador. O sistema operacional também decide quando um dispositivo de E/S pode ser usado pelo programa em execução e controla o acesso e o uso de arquivos. O próprio processador também é um recurso controlado pelo sistema operacional, que determina quanto tempo do processador deve ser dedicado à execução de cada programa de usuário. No caso de um sistema de computação com múltiplos processadores, essa decisão se estende a todos os processadores.

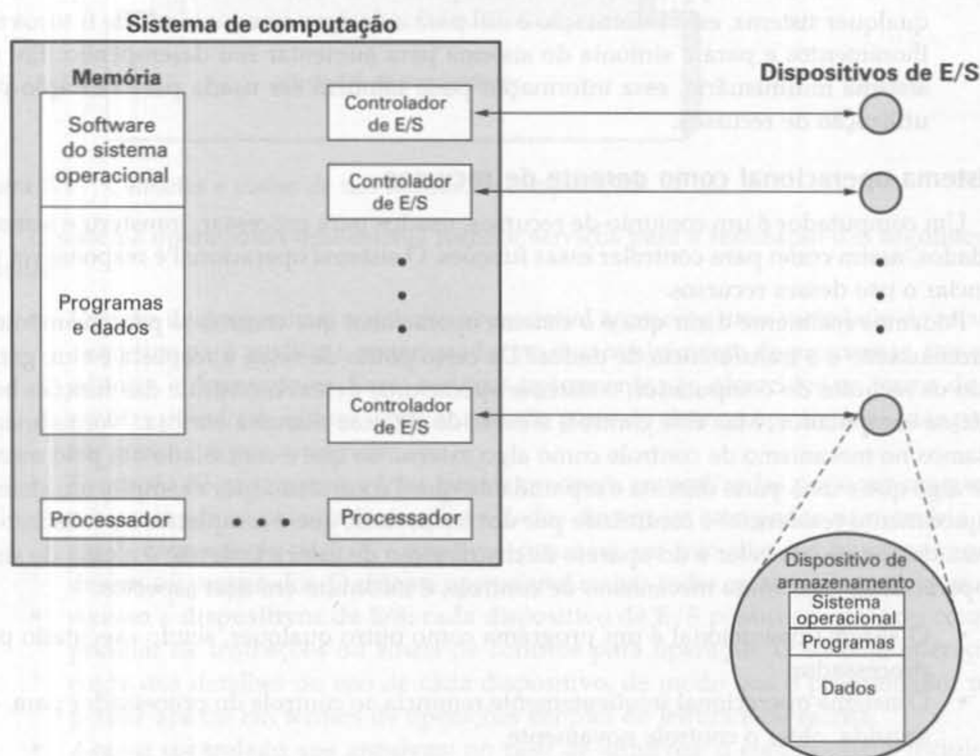


Figura 7.2 O sistema operacional como gerente de recursos.

Tipos de sistemas operacionais

Algumas características básicas diferenciam os vários tipos de sistemas operacionais. Essas características são relativas a dois aspectos independentes. O primeiro aspecto especifica se o sistema de computação é interativo ou é um sistema de processamento em lotes (*batch*). Em um sistema *interativo*, o programador/usuário interage diretamente com o com-

putador, normalmente por meio de um teclado e um monitor de vídeo, para requisitar a execução de tarefas ou efetuar transações. Além disso, ele pode, dependendo da natureza da aplicação, comunicar-se com o computador durante a execução de uma tarefa. Em um sistema de *processamento em lotes*, ocorre o oposto. Um programa de um usuário é agrupado junto com programas de outros usuários, e esse lote de programas é submetido para execução por um operador de computador. Quando a execução do programa termina, os resultados são impressos para serem entregues ao usuário. Sistemas que fazem exclusivamente processamento em lotes são raros hoje em dia. No entanto, é útil examinar brevemente esses sistemas para poder entender melhor os sistemas operacionais atuais.

Outro aspecto independente especifica se o sistema de computação emprega *multiprogramação* ou não. A multiprogramação é uma tentativa de deixar o processador ocupado o maior tempo possível, mantendo-o trabalhando em mais de um programa de cada vez. Diversos programas são simultaneamente carregados na memória, e o tempo do processador é dividido entre eles. A alternativa para esse tipo de sistema é um sistema de *monoprogramação*, que executa apenas um programa de cada vez.

Os primeiros sistemas de computação

Nos primeiros computadores, do final da década de 40 a meados da década de 50, o programador interagia diretamente com o hardware do computador: não havia sistema operacional. A execução do processador era controlada a partir de um console, que consistia em um painel de luzes, chaves de comutação e alguma forma de dispositivo de entrada (por exemplo, uma leitora de cartões) e uma impressora. Os programas, em código de processador, eram carregados através do dispositivo de entrada. Caso um erro interrompesse a execução de um programa, a condição de erro era indicada pelas luzes do painel. O programador podia então examinar os registradores e a memória principal para determinar a causa do erro. Se o programa terminasse normalmente, a saída era enviada para a impressora.

Esses primeiros sistemas de computação apresentavam dois problemas principais:

- **Escalonamento:** na maioria das instalações, usava-se uma folha de registros para reservar o tempo do processador. Tipicamente, um usuário podia reservar um dado intervalo de tempo, por exemplo, em múltiplos de meia hora. Podia ocorrer que ele reservasse, por exemplo, uma hora e a execução de seu programa terminasse em 45 minutos, o que resultava em tempo ocioso do computador. Por outro lado, um programa de usuário podia não terminar durante o tempo reservado, sendo interrompido antes de a solução de um problema ser concluída.
- **Tempo de preparação:** a execução de um único programa, denominado **tarefa** (*job*), podia envolver a carga do compilador e do programa em linguagem de alto nível (programa fonte) na memória, o armazenamento do programa compilado (programa objeto), a ligação do programa objeto a rotinas de uso comum e então a carga do programa executável na memória. Cada uma dessas etapas podia envolver a montagem ou desmontagem de fitas ou a instalação de conjuntos de cartões. Se ocorresse um erro, o usuário desafortunado tipicamente teria de reiniciar toda a seqüência de preparação. Dessa maneira, uma quantidade considerável de tempo era gasta para executar um programa.

Esse modo de operação pode ser chamado de processamento serial, refletindo o fato de que os usuários tinham acesso ao computador em série. Com o passar do tempo, foram desenvolvidas várias ferramentas de sistema para tornar o processamento em série mais eficiente. Essas ferramentas, disponíveis para todos os usuários, incluíam bibliotecas de funções de uso comum, ligadores, carregadores, depuradores de programas e rotinas de controle de E/S.

Sistemas simples de processamento em lotes

Como os primeiros processadores eram muito caros, era extremamente importante maximizar a utilização do processador. O tempo perdido em razão do escalonamento e da preparação de tarefas era inaceitável.

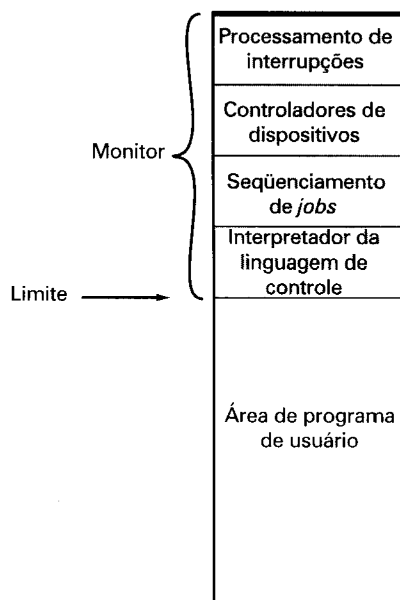


Figura 7.3 Leiaute da memória para um monitor residente.

Para melhorar a utilização do processador, foram desenvolvidos sistemas operacionais simples, de processamento em lotes. Em um sistema desse tipo, também chamado *monitor,* o usuário não tem mais acesso direto ao processador. Uma tarefa é submetida, em cartões ou fitas, a um operador de computador, que agrupa várias tarefas seqüencialmente em um lote e coloca todo esse lote em um dispositivo de entrada, a ser usado pelo monitor.

Para entender como funciona esse esquema, vamos examiná-lo sob dois pontos de vista: o do monitor e o do processador. O monitor controla a seqüência de eventos para a execução do lote de tarefas. Para isso, a maior parte do monitor deve permanecer residente na memória principal e estar disponível para a execução (Figura 7.3). Essa parte é conhecida como **monitor residente**. O restante do monitor é constituído de programas utilitários e de rotinas que implementam funções de uso comum, que são carregadas como sub-rotinas no programa de usuário no início de cada tarefa que requisite essas funções. O monitor lê as tarefas a serem executadas, uma de cada vez, a partir do dispositivo de entrada (tipicamente, uma leitora de cartões ou uma unidade de fita magnética). A tarefa lida é armazenada na área de memória

reservada para programas de usuário, e o controle é passado para essa tarefa. Quando a tarefa é completada, o controle retorna para o monitor, que imediatamente lê a próxima tarefa. Os resultados de cada tarefa são impressos para serem entregues ao usuário.

Considere agora essa mesma seqüência de eventos, do ponto de vista do processador. Em um dado momento, as instruções armazenadas na área da memória principal que contém o monitor estão sendo executadas pelo processador. Elas comandam a leitura da próxima tarefa, que é armazenada em uma outra área da memória principal. Quando essa tarefa tiver sido carregada, o processador encontrará no monitor uma instrução de desvio, que o instrui a continuar a execução a partir do início do programa de usuário. As instruções do programa de usuário são então executadas pelo processador até que a tarefa termine ou ocorra algum erro. Em qualquer um dos casos, a próxima instrução é buscada na área de memória do monitor. Dessa maneira, a frase 'o controle é passado para uma tarefa' significa simplesmente que instruções de um programa de usuário estão sendo buscadas e executadas pelo processador e a frase 'o controle é retornado ao monitor' significa que instruções do programa monitor estão sendo buscadas e executadas pelo processador.

Deve ficar claro que o monitor seleciona tarefas para execução. Um lote de tarefas é colocado em uma fila e as tarefas são executadas o mais rapidamente possível, sem que haja tempo ocioso entre elas.

E com relação ao tempo gasto na preparação de tarefas? O monitor também trata desse problema. Junto com cada tarefa, são incluídas algumas instruções, em uma **linguagem de controle de tarefas** (*job control language* — JCL), um tipo especial de linguagem de programação usada para fornecer instruções ao monitor. Um exemplo simples é a submissão de um programa FORTRAN, juntamente com os dados a serem usados pelo programa. Cada instrução ou dado de um programa FORTRAN é obtido a partir de um cartão perfurado ou de um registro de uma fita magnética. Além das instruções e dos dados do programa, a tarefa inclui instruções para controle dela, que começam com o caractere '$'. O formato geral de uma tarefa é semelhante ao que é mostrado a seguir:

```
$JOB
$FTN
•  }
•  } Instruções FORTRAN
•  }
$LOAD
$RUN
•  }
•  } Dados
•  }
$END
```

Para executar essa tarefa, o monitor lê a instrução $FTN e carrega o compilador apropriado, a partir do seu dispositivo de armazenamento de massa (normalmente uma fita magnética). O compilador traduz o programa de usuário para código objeto, que é então armazenado na memória principal ou na fita magnética. Caso o código seja armazenado na memória, a operação é conhecida como 'compilar, carregar e executar'. Caso seja armazenado na fita, será necessário incluir uma instrução $LOAD para carregá-lo na memória principal.

Essa instrução é lida pelo monitor, que retoma o controle após a compilação do programa. O monitor chama então um programa carregador, que carrega o código objeto na memória no lugar do compilador e depois transfere novamente o controle para o processador. Desse modo, é possível compartilhar uma grande parte da memória principal entre diferentes tarefas, embora apenas uma tarefa possa estar residente e ser executada de cada vez.

Podemos ver que o monitor, ou sistema operacional de processamento em lotes, é simplesmente um programa. Ele usa a capacidade do processador de buscar instruções em diferentes áreas da memória principal, para obter ou liberar o controle alternadamente. Outras características de hardware também são desejáveis:

- **Proteção de memória:** enquanto um programa de usuário está sendo executado, ele não deve alterar a área de memória que contém o monitor. Se uma tentativa de acesso à área do monitor for feita, o hardware do processador deverá detectar um erro e transferir o controle para o monitor. O monitor então interrompe e termina a tarefa, imprime uma mensagem de erro e carrega na memória a próxima tarefa.
- **Temporização:** um relógio é usado para evitar que uma única tarefa monopolize o sistema. Quando cada tarefa é iniciada, é estabelecido um determinado tempo para sua execução. Se esse tempo expirar, ocorrerá uma interrupção e o controle do processador retornará para o monitor.
- **Instruções privilegiadas:** algumas instruções podem ser definidas como privilegiadas, podendo ser executadas apenas pelo monitor. Se o processador encontrar uma instrução desse tipo durante a execução de um programa de usuário, ocorrerá uma interrupção que indica a ocorrência de um erro e o controle será transferido para o monitor. As instruções de E/S são definidas como privilegiadas, para garantir que o monitor mantenha o controle sobre todos os dispositivos de E/S. Isso evita, por exemplo, que um programa de usuário leia acidentalmente instruções de controle relativas à tarefa seguinte. Quando um programa de usuário desejar efetuar uma operação de E/S, ele deverá requisitar ao monitor que realize essa operação.
- **Interrupções:** os modelos de computadores mais antigos não possuíam capacidade de gerar e tratar interrupções. O uso de interrupções oferece ao sistema operacional maior flexibilidade para obter o controle e para renunciar ao controle do processador em favor de programas de usuários.

O tempo do processador é alternado entre a execução de programas de usuário e a execução do monitor. Isso envolve duas penalidades: parte do tempo do processador, assim como parte da memória principal, é alocada para o monitor. Essas penalidades são exemplos de sobrecarga (*overhead*). Apesar disso, um sistema operacional simples de processamento em lotes ainda melhora a utilização do computador.

Sistemas de processamento em lotes com multiprogramação

Mesmo com a execução automática de uma seqüência de tarefas, disponível em sistemas operacionais simples de processamento em lotes, o processador fica ocioso grande parte do tempo. O problema é que os dispositivos de E/S são muito lentos, se comparados ao processador. A Figura 7.4 mostra um cálculo representativo, para o caso de um programa que processa um arquivo de registros e executa, em média, cem instruções do processador por registro. Nesse exemplo, o computador gasta mais de 96% do tempo esperando que os dispo-

sitivos de E/S terminem as transferências de dados. A Figura 7.5a ilustra essa situação. Instruções são executadas pelo processador durante certo tempo, até que uma instrução de E/S seja encontrada. Nesse ponto, o processador tem de esperar até que a instrução de E/S seja concluída para prosseguir com a execução de outras instruções.

Essa ineficiência pode ser evitada. Sabemos que a capacidade da memória deve ser suficiente para armazenar o sistema operacional (monitor residente) e um programa de usuário. Suponha que o espaço de memória seja suficiente para armazenar o sistema operacional e dois programas de usuário. Nesse caso, enquanto uma tarefa aguarda a realização de uma operação de E/S, o processador pode executar outra tarefa, que provavelmente não estará aguardando E/S (Figura 7.5b). Além disso, a capacidade da memória pode ser aumentada ainda mais, para conter três, quatro ou mais programas, sendo o processamento alternado entre todos eles (Figura 7.5c). Esse processo é conhecido como **multiprogramação** e constitui uma característica fundamental dos sistemas operacionais modernos.

Ler um registro	0,0015 segundo
Executar 100 instruções	0,0001 segundo
Escrever um registro	0,0015 segundo
TOTAL	0,0031 segundo

$$\text{Porcentagem de utilização da CPU} = \frac{0{,}0001}{0{,}0031} = 0{,}032 = 3{,}2\%$$

Figura 7.4 Exemplo de utilização de sistema.

A vantagem da multiprogramação pode ser mostrada por meio do seguinte exemplo. Considere um computador com 256K palavras de memória disponível (não usada pelo sistema operacional), um disco, um terminal e uma impressora. Três programas, P1, P2 e P3, cujos atributos são apresentados na Tabela 7.1, são submetidos para a execução ao mesmo tempo. Suponhamos que P2 e P3 requeiram pouco tempo do processador e que P3 use o disco e a impressora continuamente. Em um ambiente simples de processamento em lotes, essas tarefas seriam executadas em seqüência. Dessa maneira, P1 seria completado em 5 minutos. P2 teria de esperar 5 minutos para iniciar sua execução e terminaria 15 minutos depois disso. A execução de P3 seria iniciada após 20 minutos, sendo completada 30 minutos depois de P3 ter sido submetido. A utilização média dos recursos do sistema, o número de tarefas executadas por unidade de tempo e os tempos de resposta de cada tarefa são mostrados na Tabela 7.2, na coluna referente à monoprogramação. A utilização de cada recurso do sistema é mostrada na Figura 7.6. É evidente a enorme subutilização de todos os recursos, quando computada ao longo dos 30 minutos requeridos para a execução das três tarefas.

Suponha agora que as tarefas são executadas concorrentemente em um sistema operacional com multiprogramação. Como existe baixa contenção de recursos entre as tarefas, cada tarefa pode ser executada quase no tempo mínimo, enquanto coexiste com as demais tarefas no computador (supondo que o tempo de processador alocado para P2 e P3 seja suficiente para que se possam manter ativas suas operações de entrada e de saída). A tarefa P1 ainda precisará de 5 minutos de tempo de processador para ser completada. Entretanto, ao final desse tempo, um terço de P2 já terá sido executado, assim como a metade de P3. As três tarefas são completadas em 15 minutos. O ganho obtido torna-se evidente quando se observa a coluna da Tabela 7.2, referente à multiprogramação, que é obtida a partir do histograma mostrado na Figura 7.7.

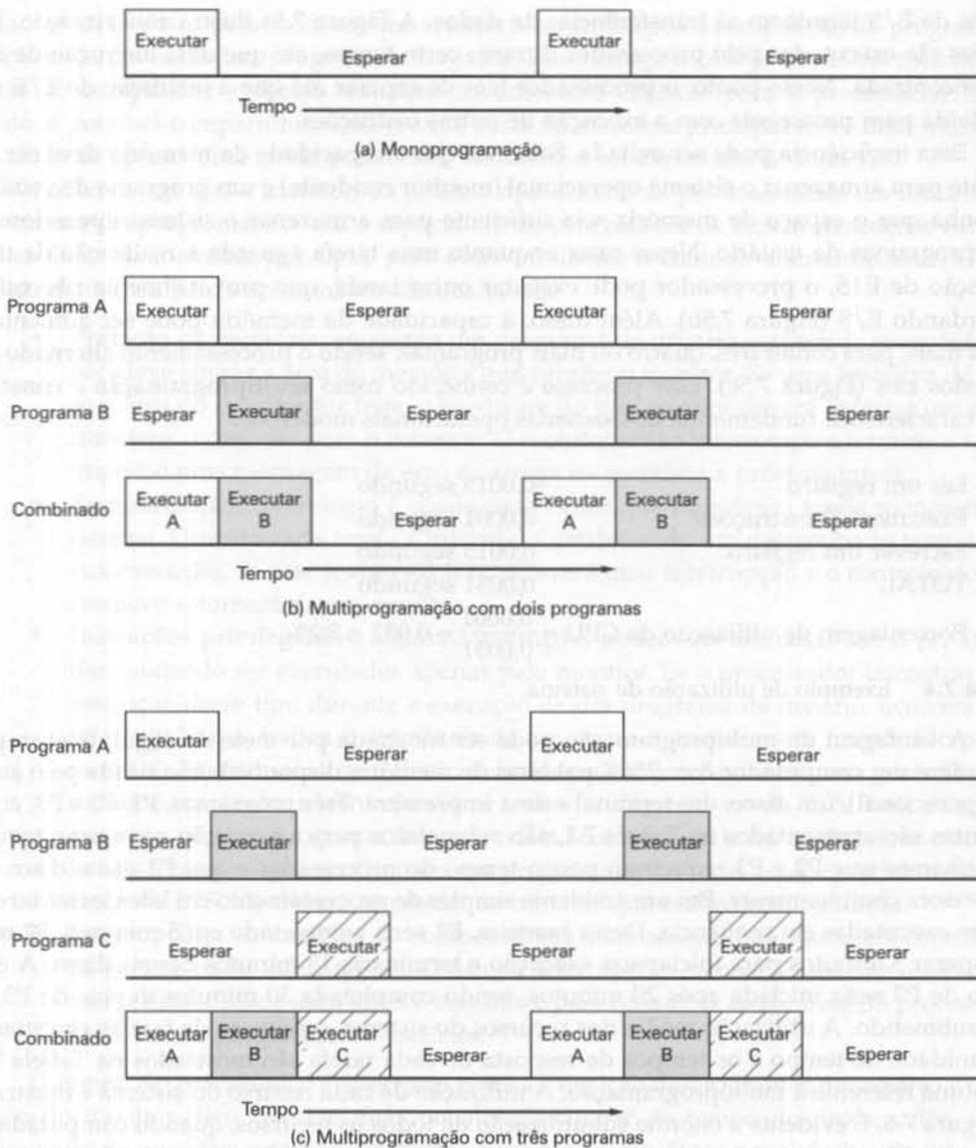


Figura 7.5 Exemplo de multiprogramação.

Assim como um sistema de processamento em lote simples, um sistema com multiprogramação depende de certas características do hardware do computador. A característica adicional fundamental para a multiprogramação é o hardware que suporta as interrupções de E/S e o acesso direto à memória. Com a E/S dirigida por interrupção ou o acesso direto à memória, o processador pode submeter uma operação de E/S de uma dada tarefa e prosseguir executando outra tarefa enquanto a E/S é efetuada pelo controlador do dispositivo. Quando a E/S é completada, o processador é interrompido e o controle é transferido para uma rotina de tratamento de interrupção do sistema operacional, o qual passa, então, o controle para outra tarefa.

Tabela 7.1 Exemplos de atributos de execução de programas

	P1	P2	P3
Tipo de tarefa	Computação intensiva	E/S intensiva	E/S intensiva
Duração	5 min	15 min	10 min
Memória requerida	50K	100K	80K
Usa disco?	Não	Não	Sim
Usa terminal?	Não	Sim	Não
Usa impressora?	Não	Não	Sim

Os sistemas operacionais com multiprogramação são razoavelmente sofisticados, se comparados a sistemas com **monoprogramação**. Para que várias tarefas possam estar prontas para execução, elas devem ser mantidas na memória principal, o que requer alguma forma de **gerenciamento de memória**. Além disso, se diversas tarefas estiverem prontas para execução, o processador deve decidir qual delas deve ser executada a cada instante, o que requer algum algoritmo de escalonamento de tarefas. Esses conceitos são discutidos mais adiante, ainda neste capítulo.

Tabela 7.2 Efeito da multiprogramação na utilização de recursos.

	Monoprogramação	Multiprogramação
Uso de processador	17%	33%
Uso de memória	30%	67%
Uso de disco	33%	67%
Uso de impressora	33%	67%
Tempo decorrido	30 min	15 min
Taxa de execução de tarefas	6 tarefas/h	12 tarefas/h
Tempo de resposta médio	18 min	10 min

Sistemas de tempo compartilhado

Com o uso de multiprogramação, o processamento em lotes pode ser bastante eficiente. Entretanto, para muitas tarefas, é desejável permitir uma interação direta do usuário com o computador. De fato, para algumas tarefas, tais como o processamento de transações, o modo interativo é essencial.

Atualmente, a necessidade de computação interativa é muitas vezes atendida pelo uso de microcomputadores dedicados. Essa opção não era disponível nos anos 60, quando boa parte dos computadores era grande e cara. Para suprir essa necessidade, foi desenvolvido o conceito de compartilhamento do tempo de processador.

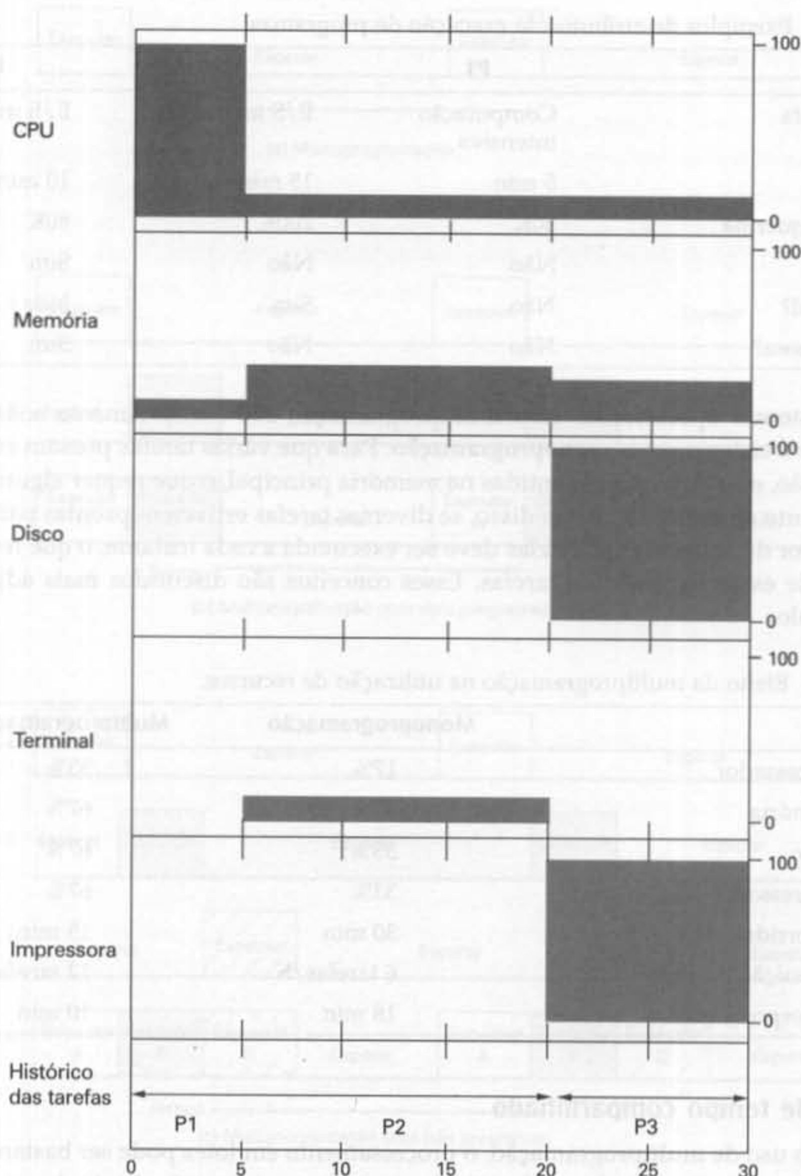


Figura 7.6 Histograma de utilização de recursos em um sistema com monoprogramação.

Além de possibilitar ao processador executar um lote de tarefas de uma só vez, a multiprogramação também pode ser empregada para executar várias tarefas interativas. Nesse último caso, a técnica é conhecida como compartilhamento de tempo, porque o tempo do processador é dividido entre vários usuários. Em um sistema de tempo compartilhado, diversos usuários usam o sistema simultaneamente, por meio de terminais, tendo o sistema operacional como responsável por intercalar a execução dos programas de usuário, executando cada tarefa por um determinado intervalo de tempo de cada vez. Dessa maneira, se n usuários requisitarem serviços ao mesmo tempo, cada usuário terá a visão de um sistema com $1/n$ da

velocidade efetiva do computador, em média, desconsiderando o tempo consumido pelo sistema operacional. Entretanto, como o tempo de reação do ser humano é relativamente lento, o tempo de resposta de um sistema projetado adequadamente deve ser semelhante ao tempo de resposta de computador dedicado.

Tanto sistemas de processamento em lotes com multiprogramação quanto sistemas de tempo compartilhado utilizam a multiprogramação. As diferenças-chave entre esses dois tipos de sistemas são mostradas na Tabela 7.3.

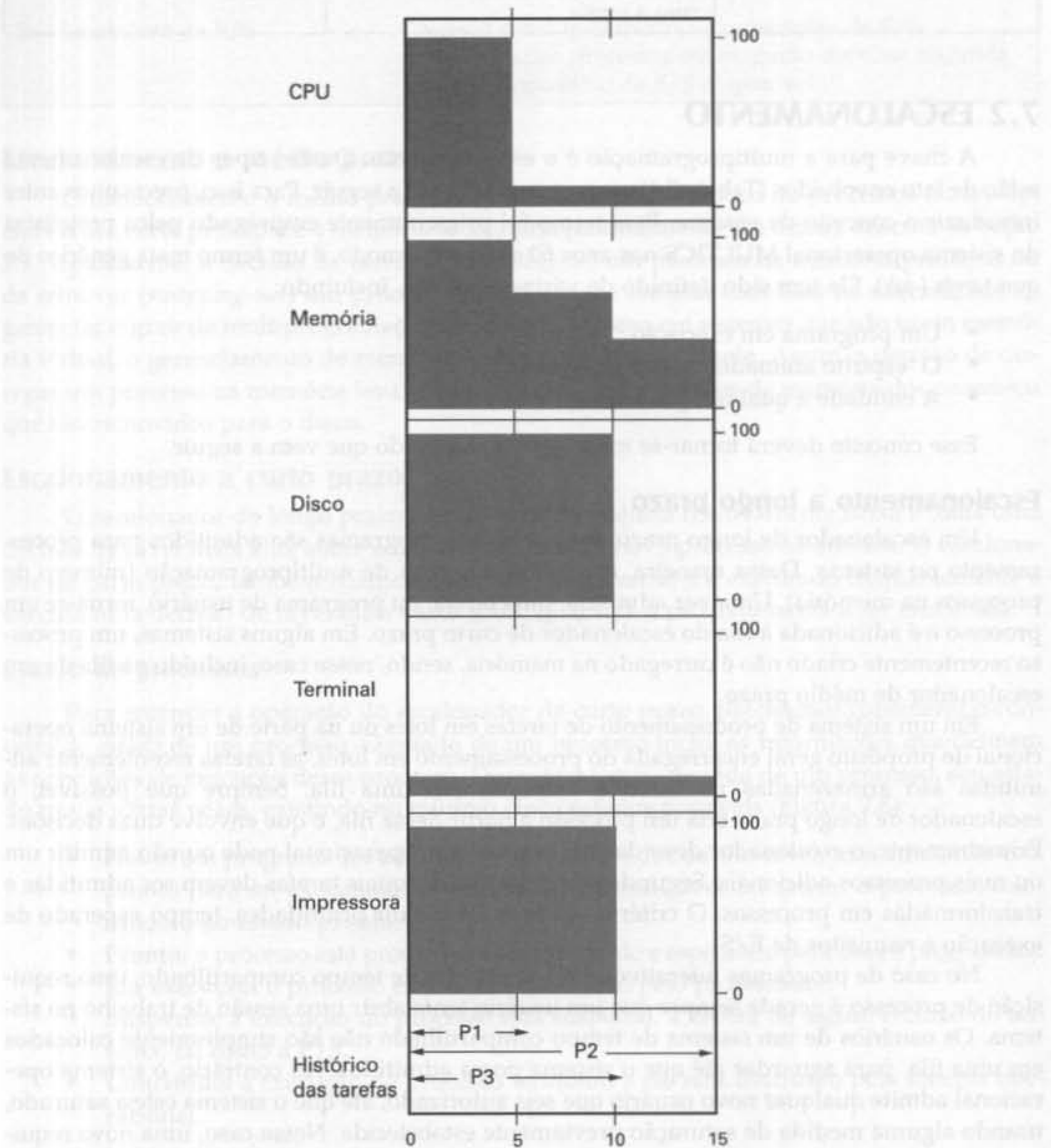


Figura 7.7 Histograma de utilização de recursos em um sistema com multiprogramação.

Tabela 7.3 Processamento de tarefas em lotes com multiprogramação *versus* compartilhamento de tempo

	Processamento de tarefas em lotes com multiprogramação	Compartilhamento de tempo
Objetivo principal	Maximizar o uso do processador	Minimizar o tempo de resposta
Fonte de instruções para o sistema operacional	Instruções de linguagem de controle de tarefas fornecidas com a tarefa	Comandos enviados pelo terminal

7.2 ESCALONAMENTO

A chave para a multiprogramação é o escalonamento. Quatro tipos de escalonamento estão de fato envolvidos (Tabela 7.4), como apresentamos a seguir. Para isso, precisamos antes introduzir o conceito de *processo*. Esse termo foi primeiramente empregado pelos projetistas do sistema operacional MULTICS nos anos 60 e, de certo modo, é um termo mais genérico do que *tarefa* (*job*). Ele tem sido definido de várias maneiras, incluindo:

- Um programa em execução.
- O 'espírito animado' de um programa.
- A entidade à qual um processador é alocado.

Esse conceito deverá tornar-se mais claro ao longo do que vem a seguir.

Escalonamento a longo prazo

Um escalonador de longo prazo determina que programas são admitidos para processamento no sistema. Dessa maneira, ele controla o grau de multiprogramação (número de processos na memória). Uma vez admitida, uma tarefa, ou programa de usuário, torna-se um processo e é adicionada à fila do escalonador de curto prazo. Em alguns sistemas, um processo recentemente criado não é carregado na memória, sendo, nesse caso, incluído na fila de um escalonador de médio prazo.

Em um sistema de processamento de tarefas em lotes ou na parte de um sistema operacional de propósito geral encarregada do processamento em lotes, as tarefas recentemente admitidas são armazenadas no disco e mantidas em uma fila. Sempre que possível, o escalonador de longo prazo cria um processo a partir dessa fila, o que envolve duas decisões. Primeiramente, o escalonador deve decidir se o sistema operacional pode ou não admitir um ou mais processos adicionais. Segundo, ele deve decidir quais tarefas devem ser admitidas e transformadas em processos. O critério usado pode incluir prioridades, tempo esperado de execução e requisitos de E/S.

No caso de programas interativos em um sistema de tempo compartilhado, uma requisição de processo é gerada sempre que um usuário tenta abrir uma sessão de trabalho no sistema. Os usuários de um sistema de tempo compartilhado não são simplesmente colocados em uma fila, para aguardar até que o sistema possa admiti-los. Ao contrário, o sistema operacional admite qualquer novo usuário que seja autorizado, até que o sistema esteja saturado, usando alguma medida de saturação previamente estabelecida. Nesse caso, uma nova requisição para abertura de sessão resulta em uma mensagem indicando que o sistema está saturado e que o usuário deve tentar uma nova conexão mais tarde.

Tabela 7.4 Tipos de escalonamento

Escalonamento a longo prazo	Decisão de acrescentar um novo processo ao conjunto de processos a serem executados.
Escalonamento a médio prazo	Decisão de acrescentar um processo ao conjunto de processos que estão parcial ou completamente carregados na memória principal.
Escalonamento a curto prazo	Decisão sobre qual dos processos disponíveis na memória será executado pelo processador.
Escalonamento de E/S	Decisão sobre qual dentre as requisições de E/S pendentes dos processos em execução deve ser atendida por um dispositivo de E/S disponível.

Escalonamento a médio prazo

O escalonamento a médio prazo faz parte da função de troca de processos (*swapping*) entre a memória principal e a memória secundária (normalmente um disco), descrita na Seção 7.3. Tipicamente, a decisão de carregar (*swapping-in*) um processo na memória principal ou de remover (*swapping-out*) um processo para o disco é tomada com base na necessidade de gerenciar o grau de multiprogramação do sistema. Mesmo em sistemas que não usam memória virtual, o gerenciamento de memória é uma questão importante. Assim, a decisão de carregar um processo na memória leva em consideração os requisitos de memória dos processos que são removidos para o disco.

Escalonamento a curto prazo

O escalonador de longo prazo executa com freqüência relativamente baixa e toma uma decisão de nível mais alto, sobre adicionar ou não um novo processo ao sistema. O escalonador de curto prazo, também chamado *despachante* (*dispatcher*), é executado freqüentemente e executa uma decisão de nível mais baixo sobre qual será a próxima tarefa a ser executada.

Estado de processos

Para entender a operação do escalonador de curto prazo, precisamos considerar o conceito de *estado* de um processo. O estado de um processo inclui as informações que definem as condições de execução desse processo. Durante o tempo de vida de um processo, seu estado muda várias vezes, existindo no mínimo cinco estados possíveis (Figura 7.8):

- **Novo:** um programa foi admitido pelo escalonador de alto nível, mas ainda não está pronto para ser executado. O sistema operacional deve inicializar o processo, colocando-o no estado pronto.
- **Pronto:** o processo está pronto para ser executado e esperando para usar o processador.
- **Em execução:** o processo está sendo executado pelo processador.
- **Suspenso:** a execução do processo está suspensa, à espera de algum recurso do sistema, tal como a E/S.
- **Concluído:** a execução do processo terminou e ele será destruído pelo sistema operacional.



Figura 7.8 Modelo de processo com cinco estados.

O sistema operacional deve manter, para cada processo, informações sobre seu estado e outras informações necessárias para sua execução. Para isso, cada processo é representado no sistema operacional por um *bloco de controle de processos* (Figura 7.9), que contém tipicamente as seguintes informações:

- **Identificador:** cada processo tem um identificador distinto.
- **Estado:** estado atual do processo (novo, pronto etc.).
- **Prioridade:** nível de prioridade relativa.
- **Contador de programa:** endereço da próxima instrução a ser executada.
- **Limites de memória:** endereços inicial e final da área de memória ocupada pelo processo.
- **Informações de contexto:** dados contidos nos registradores do processador enquanto o processo está sendo executado, a serem discutidos na Parte III. Por enquanto, é suficiente dizer que esses dados representam o "contexto" do processo. Os dados de contexto, juntamente com o contador do programa, são guardados pelo sistema operacional quando a execução do processo é suspensa. Eles são recuperados quando o processador retoma a execução do processo.
- **Informação de estado de E/S:** inclui requisitos de E/S pendentes, dispositivos de E/S (por exemplo, unidades de fita) alocados ao processo, lista dos arquivos alocados ao processo e assim por diante.
- **Informação de contabilidade:** pode incluir a quantidade de tempo do processador e o tempo total já usados pelo processo, limites de tempo de execução, contabilização de uso de recursos e assim por diante.

Quando o escalonador admite uma nova tarefa ou uma nova requisição de usuário, ele cria um bloco de controle de processos em branco e coloca o processo no estado *novo*. Depois que o sistema tiver preenchido adequadamente as informações do bloco de controle, o processo será colocado no estado *pronto*.

Identificador
Estado
Prioridade
Contador de programa
Limites de memória
Informações de contexto
Informação de estado de E/S
Informação de contabilidade
⋮

Figura 7.9 Bloco de controle de processos.

Técnicas de escalonamento

Para entender como o sistema operacional gerencia o escalonamento das várias tarefas presentes na memória, considere o exemplo simples da Figura 7.10. A figura mostra como a memória principal está repartida em um determinado instante do tempo. O núcleo do sistema operacional é sempre residente na memória. Além disso, existe certo número de processos ativos, incluindo os processos *A* e *B*, cada qual carregado em uma área da memória.

Em um determinado instante, o processo *A* está sendo executado. Instruções do programa contidas na área de memória de *A* (correspondentes ao processo *A*) estão sendo executadas pelo processador. Mais tarde, o processador interrompe a execução de instruções em *A* e começa a executar as instruções da área correspondente ao sistema operacional. Isso ocorre em virtude de uma das três razões a seguir:

1. O processo A efetua uma chamada ao sistema operacional (por exemplo, um requisito de E/S). A execução de *A* é então suspensa até que essa chamada seja atendida pelo sistema operacional.
2. A execução do processo *A* gera uma *interrupção*. Uma interrupção é um sinal gerado pelo hardware para o processador. Ao detectar esse sinal, o processador interrompe a execução de *A* e transfere o controle para a rotina de tratamento de interrupções do sistema operacional. Diversos eventos relacionados ao processo *A* podem causar uma interrupção. Um exemplo é um erro, como uma tentativa de executar uma instrução privilegiada. Um outro é o término do intervalo de tempo alocado para a execução do processo. Para evitar que o processador seja monopolizado por um processo, ele é alocado a cada processo apenas por um curto período de tempo.
3. Algum evento não relacionado ao processo *A* causa uma interrupção. Um exemplo é a conclusão de uma operação de E/S.

Qualquer um desses eventos resulta no seguinte. O processador armazena, para uso posterior, o contador de programa e as informações de contexto do processo corrente (proces-

so *A*) no bloco de controle do processo *A* e o sistema operacional começa então a ser executado. O sistema operacional pode realizar algum trabalho, como iniciar uma operação de E/S. Em seguida, o escalonador de curto prazo do sistema operacional é executado, para decidir qual processo deve ser executado a seguir. Nesse exemplo, o processo *B* é escolhido. O sistema operacional instrui o processador para recuperar as informações de contexto do processo *B* e retomar sua execução a partir do ponto em que foi interrompida anteriormente.

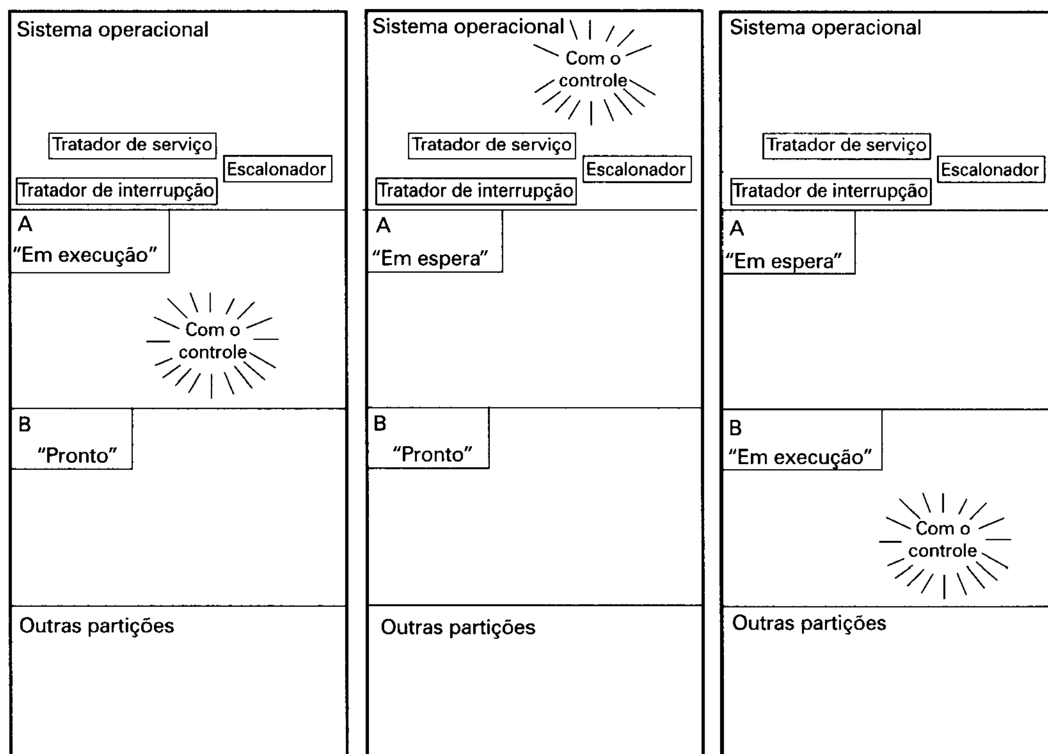


Figura 7.10 Exemplo de escalonamento.

Esse exemplo simples destaca o funcionamento básico do escalonador de curto prazo. A Figura 7.11 mostra os principais elementos do sistema operacional envolvidos na multiprogramação e no escalonamento de processos. Caso ocorra uma interrupção, o controle é passado para a rotina de tratamento de interrupções do sistema operacional; no caso de ocorrer uma chamada ao sistema, quem recebe o controle é a rotina de tratamento da chamada correspondente. Uma vez tratada a interrupção ou a chamada ao sistema, o escalonador de curto prazo é chamado para escolher um processo para a execução.

Para cumprir sua tarefa, o sistema operacional utiliza várias filas. Cada fila consiste simplesmente em uma lista de processos que aguardam algum recurso. A *fila de longo prazo* contém tarefas que estão aguardando para usar o sistema. Quando as condições permitirem, o escalonador de alto nível alocará a memória e criará um novo processo. A *fila de curto prazo* contém todos os processos prontos para a execução. Qualquer um desses processos pode ser o próximo a ser executado pelo processador. Decidir qual será o próximo processo a usar o

processador fica a cargo do escalonador de curto prazo. Geralmente, isso é feito com um algoritmo de alocação circular (*round-robin*), alocando a cada processo um determinado período de tempo de cada vez. Níveis de prioridade podem também ser usados. Finalmente, existe uma *fila de E/S* para cada dispositivo de E/S, uma vez que mais de um processo pode requisitar o uso de um mesmo dispositivo. Todos os processos que estão esperando para usar um dispositivo são colocados na fila correspondente a esse dispositivo.

A Figura 7.12 mostra como é feita a execução de processos no computador, sob controle do sistema operacional. Cada requisição de processo (execução de tarefa de um lote de tarefas ou tarefa interativa) é colocada na fila de longo prazo. Quando os recursos necessários se tornam disponíveis, a requisição é atendida, sendo criado um processo no estado pronto, que é colocado na fila de curto prazo. O processador alterna entre a execução de instruções do sistema operacional e a de processos de usuário. Enquanto detém o controle do processador, o sistema operacional decide qual dos processos da fila de curto prazo será executado a seguir. Quando o sistema operacional terminar a execução de suas tarefas imediatas, ele transferirá o controle do processador para o processo escolhido.

Como foi dito anteriormente, o processo que está sendo executado pode ser suspenso por vários motivos. Se ele for suspenso porque requisitou a realização de uma operação de E/S, ele será colocado na fila de E/S adequada. Se for suspenso porque o período de tempo alocado para sua execução expirou ou porque o sistema operacional tem de atender algum serviço urgente, ele será colocado no estado pronto e acrescentado à fila de curto prazo.

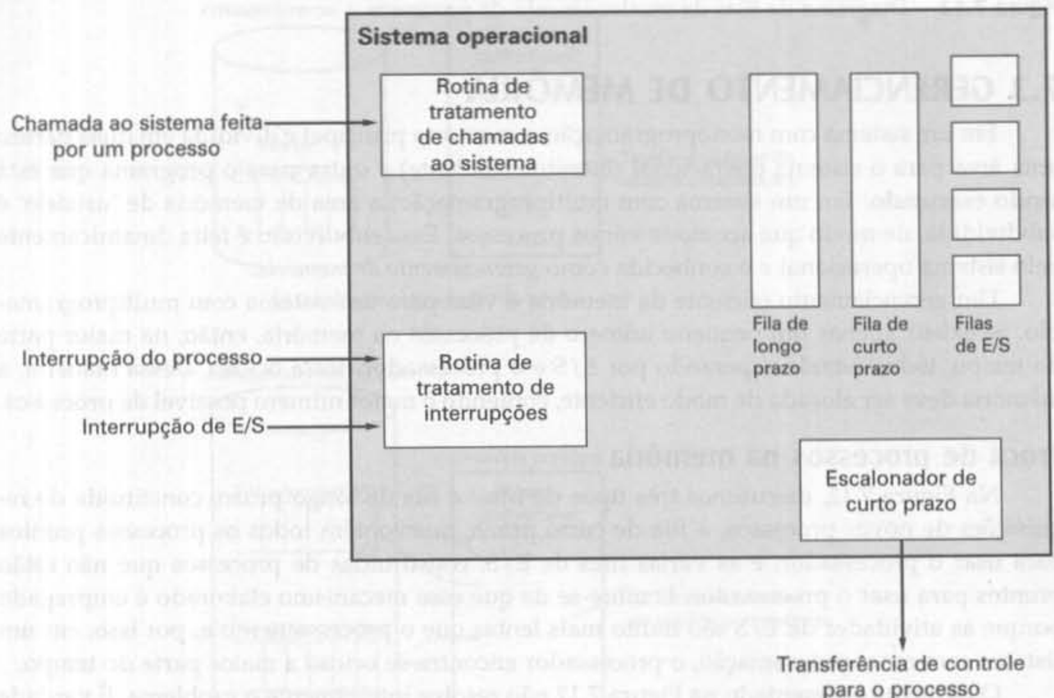


Figura 7.11 Elementos-chave de um sistema operacional com multiprogramação.

Finalmente, mencionamos que o sistema operacional gerencia, também, as filas de E/S. Quando uma operação de E/S é completada, o sistema operacional remove o processo atendido da fila de E/S, colocando-o na fila de curto prazo. Ele então seleciona um dos processos em espera na fila do dispositivo (se houver) e envia um sinal ao dispositivo para que ele atenda à requisição desse processo.

Figura 7.12 Diagrama de filas de escalonamento de processos.

7.3 GERENCIAMENTO DE MEMÓRIA

Em um sistema com monoprogramação, a memória principal é dividida em duas partes: uma área para o sistema operacional (monitor residente) e outra para o programa que está sendo executado. Em um sistema com multiprogramação, a área de memória de 'usuário' é subdividida, de modo que acomode vários processos. Essa subdivisão é feita dinamicamente pelo sistema operacional e é conhecida como *gerenciamento de memória*.

Um gerenciamento eficiente da memória é vital para um sistema com multiprogramação. Se existir apenas um pequeno número de processos na memória, então, na maior parte do tempo, todos estarão esperando por E/S e o processador ficará ocioso. Dessa maneira, a memória deve ser alocada de modo eficiente, contendo o maior número possível de processos.

Troca de processos na memória

Na Figura 7.12, discutimos três tipos de filas: a fila de longo prazo, constituída de requisições de novos processos, a fila de curto prazo, que contém todos os processos prontos para usar o processador, e as várias filas de E/S, constituídas de processos que não estão prontos para usar o processador. Lembre-se de que esse mecanismo elaborado é empregado porque as atividades de E/S são muito mais lentas que o processamento e, por isso, em um sistema com monoprogramação, o processador encontra-se ocioso a maior parte do tempo.

O esquema apresentado na Figura 7.12 não resolve inteiramente o problema. É verdade que, nesse caso, diversos processos ficam armazenados na memória, e o uso do processador é transferido para outro processo enquanto um processo aguarda por E/S. Entretanto, o processador é tão mais rápido que os dispositivos de E/S que é comum uma situação em que

todos os processos carregados na memória estejam aguardando por E/S. Portanto, mesmo com a multiprogramação, um processador pode ficar ocioso a maior parte do tempo.

O que fazer então? A memória principal poderia ser aumentada, de modo a acomodar maior número de processos. Mas essa abordagem apresenta duas falhas. A primeira é que, mesmo hoje em dia, a memória principal é cara e a segunda refere-se à capacidade de a memória requerida pelos programas ter crescido tão rapidamente quanto o custo da memória tem diminuído. Assim, memórias maiores têm resultado em processos maiores, e não em maior número de processos.

Outra solução é a *troca de processos* (*swapping*) na memória, representada na Figura 7.13. A fila de requisições de processos de longo prazo é tipicamente armazenada em disco. Os processos são trazidos para a memória principal, um de cada vez, assim que haja espaço disponível. Quando um processo termina, ele é novamente transferido para o disco. Pode acontecer de nenhum dos processos carregados na memória estar pronto para a execução (por exemplo, todos podem estar aguardando a realização de uma operação de E/S). Em vez de permanecer ocioso, o processador *troca* um desses processos, retirando-o da memória para uma *fila intermediária* no disco. Essa fila contém processos já criados, mas que foram temporariamente retirados da memória. O sistema operacional, então, carrega na memória um outro processo dessa fila intermediária ou atende a uma nova requisição de processo da fila de longo prazo. A execução, então, continua com o processo recém-carregado na memória.

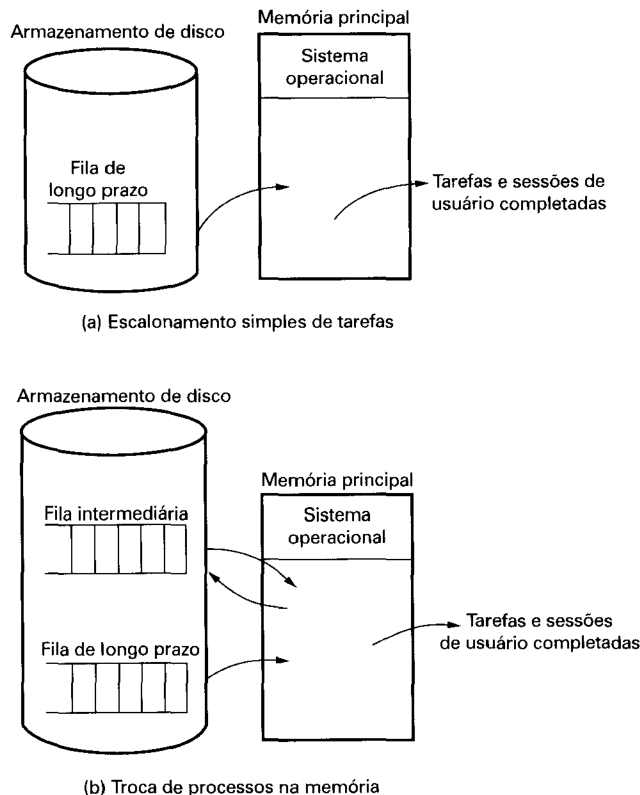


Figura 7.13 Troca de processos na memória.

Como a troca de processos na memória é uma operação de E/S, existe o risco de essa estratégia piorar ainda mais o problema, em vez de ser uma solução. No entanto, como a E/S em disco geralmente é mais rápida que nos demais dispositivos do sistema (por exemplo, se comparada com a E/S em fita magnética ou impressora), a troca de processos geralmente melhora o desempenho. Um esquema mais sofisticado, envolvendo a memória virtual, melhora ainda mais o desempenho na troca de processos na memória. Isso será discutido brevemente a seguir. Para isso, precisamos antes introduzir os conceitos de partição de memória e paginação.

Partição de memória

O esquema mais simples para dividir a memória disponível entre os processos é usar *partições de tamanho fixo*, como mostrado na Figura 7.14. Note que, embora as partições sejam de tamanho fixo, elas não precisam ter o mesmo tamanho. Quando um processo é trazido para a memória, ele é carregado na menor partição disponível capaz de contê-lo.

Mesmo o uso de partições fixas de tamanhos desiguais traz certo desperdício de memória. Na maioria dos casos, um processo não requer uma área de memória exatamente igual ao tamanho de uma partição disponível. Por exemplo, na Figura 7.14b, um processo que requisita 3 Mbytes de memória pode ser carregado na partição de 4 Mbytes, desperdiçando 1 Mbyte que poderia ser usado por outro processo.

Figura 7.14 Exemplo de partições fixas de uma memória de 64 Mbytes.

Uma abordagem mais eficiente é usar *partições de tamanho variável*. A área de memória alocada a cada processo é exatamente do tamanho requerido. A Figura 7.15 mostra um exemplo em que a memória principal tem um tamanho total de 1 Mbyte. Inicialmente, a memória contém apenas o sistema operacional (a). Os primeiros três processos são carregados a partir do final da área do sistema operacional, ocupando espaço suficiente para cada processo (b, c, d). Resta um "buraco" no final da memória, de tamanho muito pequeno para conter um quarto processo. Em um dado instante, nenhum desses processos está pronto. O sistema operacional então retira da memória o processo 2 (e), deixando espaço suficiente para carregar um novo processo, o processo 4 (f). Como o processo 4 é menor que o processo 2, outro buraco pequeno é criado. Mais tarde, novamente nenhum dos processos presentes na memória principal está pronto, mas o processo 2, que está no estado pronto/suspenso, se encontra disponível. Como não há espaço suficiente para o processo 2, o sistema operacional retira o processo 1 (g), trazendo de volta o processo 2 (h) para a memória. Como mostra esse exemplo, essa estratégia começa bem, mas, depois de algum tempo, leva a uma situação em que existem diversos buracos pequenos na memória. A memória torna-se mais e mais fragmentada e a sua utilização diminui. Uma técnica usada para evitar esse problema é a *compactação*: de tempos em tempos, o sistema operacional troca os processos na memória de lugar, agrupando todas as áreas de memória livre em um único bloco. Esse procedimento consome parte do tempo útil do processador.

Antes de considerar algumas maneiras de evitar as desvantagens do método de partição de memória, temos de esclarecer um ponto em particular. Observando a Figura 7.15, fica claro que um processo não precisa ser carregado sempre na mesma posição, toda vez que é trazido para a memória. Além disso, se a compactação for usada, um processo poderá ser trocado de posição, mesmo enquanto estiver na memória principal. Um processo carregado na memória principal é constituído de instruções e dados. As instruções contêm dois tipos de endereços de memória:

- Endereços de dados
- Endereços de instruções, usados em instruções de desvio

Esses endereços não são fixos, podendo mudar toda vez que o processo for trazido para a memória principal. Para isso, é feita uma distinção entre endereços lógicos e endereços físicos. Um *endereço lógico* corresponde a uma posição relativa ao início do programa. As instruções do programa contêm apenas endereços lógicos. Um *endereço físico* designa uma posição na memória principal. Quando o processador executa um processo, ele automaticamente converte um endereço lógico para um endereço físico, somando ao endereço lógico a posição inicial da área de memória do processo, conhecida como seu *endereço-base*. Isso é mais um exemplo de característica do hardware do processador, projetada para atender a uma necessidade do sistema operacional. A natureza exata dessa característica de hardware depende da estratégia de gerenciamento de memória em uso. Diversos exemplos serão vistos no decorrer deste capítulo.

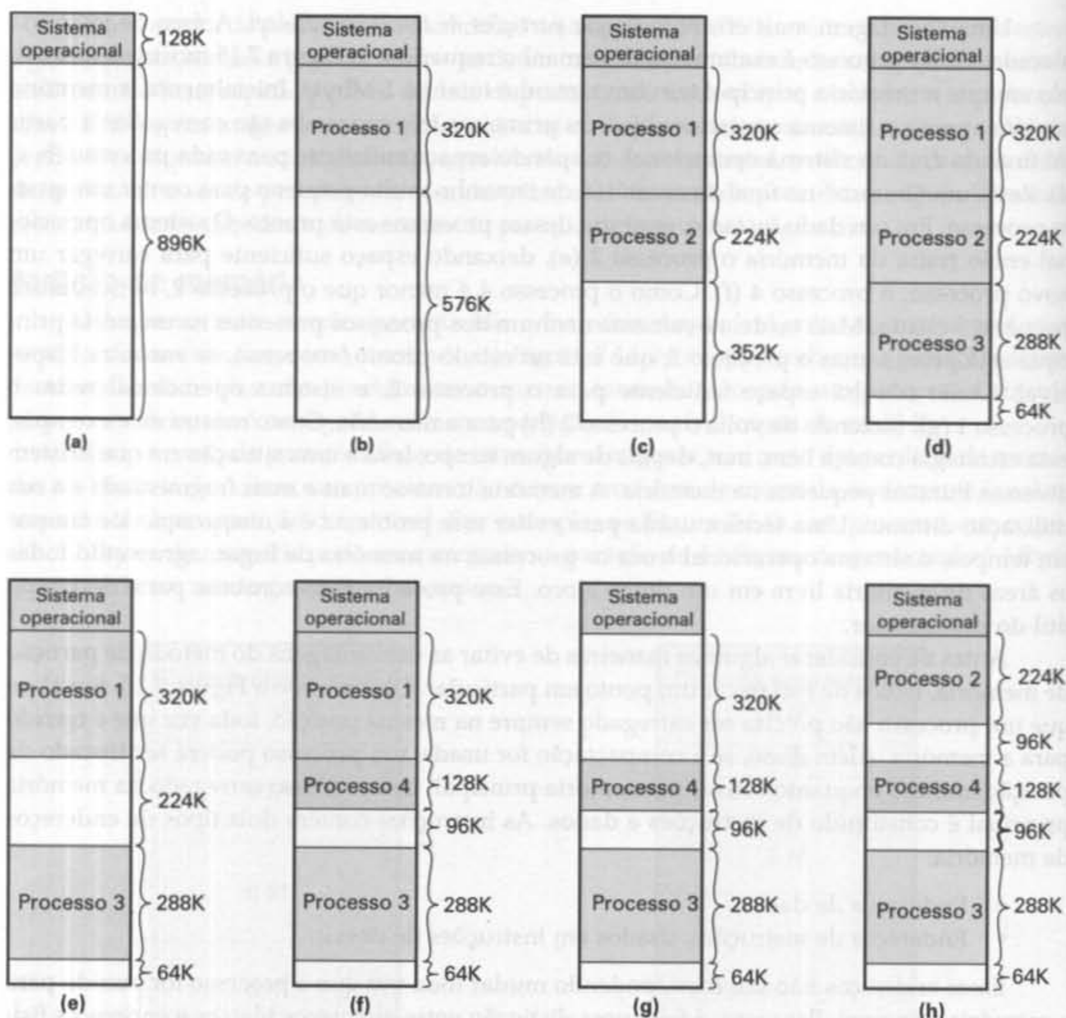


Figura 7.15 Efeito da partição dinâmica de memória.

Paginação de memória

O uso de partições, de tamanho fixo ou de tamanho variável, ainda resulta em uma utilização ineficiente da memória. Suponha, contudo, que a memória seja dividida em partes iguais de tamanho fixo, relativamente pequenas, e que cada processo seja também dividido em pequenos pedaços, com determinado tamanho fixo. Esses pedaços, conhecidos como *páginas*, podem ser alocados em partes disponíveis da memória, conhecidas como *blocos* (*frames*). O espaço desperdiçado na memória com a carga de um processo é, então, apenas uma fração do último bloco alocado ao processo.

A Figura 7.16 mostra um exemplo do uso de páginas e blocos. Em um determinado instante, alguns blocos da memória estão em uso e outros estão livres. Uma lista dos blocos livres é mantida pelo sistema operacional. O processo *A*, armazenado em disco, é constituído de

quatro páginas. Quando ele deve ser carregado na memória, o sistema operacional procura quatro blocos livres e carrega as quatro páginas do processo *A* nesses blocos.

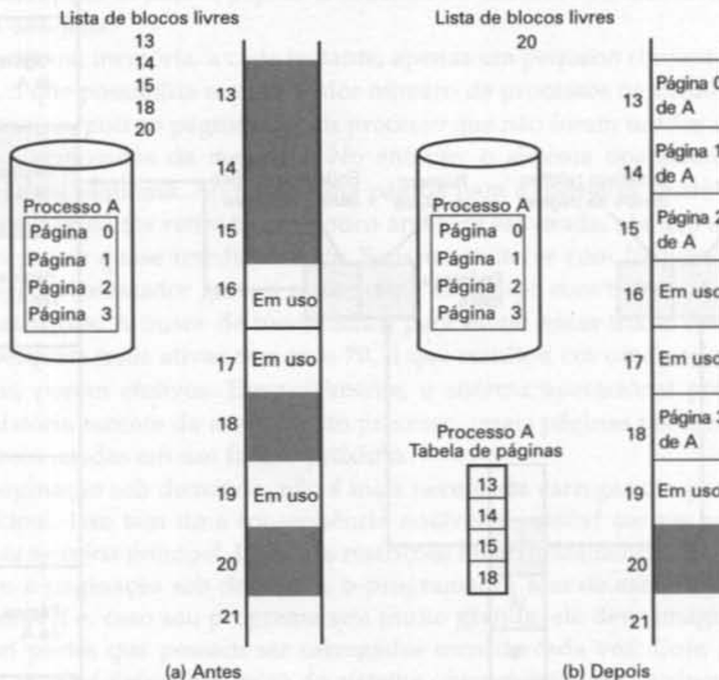


Figura 7.16 Alocação de blocos livres.

Suponha agora, como nesse exemplo, que não há um número suficiente de blocos contíguos para alocar ao processo. Isso impediria que o sistema operacional carregasse na memória o processo *A*? A resposta é não, porque, mais uma vez, pode ser usado o conceito de endereço lógico. Um simples endereço-base não é mais suficiente. Em vez disso, o sistema operacional mantém uma *tabela de páginas* para cada processo, que contém o endereço do bloco de cada página do processo. Cada endereço lógico no programa é composto de um número de página e um endereço relativo dentro da página. Lembre-se de que, no caso de partição simples, um endereço lógico é um endereço relativo ao começo do programa; o processador traduz esse endereço para um endereço físico. No esquema de paginação, a tradução de endereço lógico para endereço físico também é feita pelo hardware do processador. Para isso, o processador usa a tabela de páginas do processo corrente. Dado um endereço lógico (número de página, endereço relativo), o processador usa a tabela de páginas para produzir um endereço físico (número de bloco, endereço relativo). A Figura 7.17 mostra um exemplo.

Essa abordagem resolve o problema descrito anteriormente. A memória principal é dividida em diversos blocos pequenos de mesmo tamanho. Cada processo é dividido em páginas de tamanho igual ao de um bloco: processos menores requerem poucas páginas e processos maiores, mais páginas. Quando um processo é trazido para a memória, suas páginas são carregadas em blocos disponíveis e uma tabela de páginas é preparada.

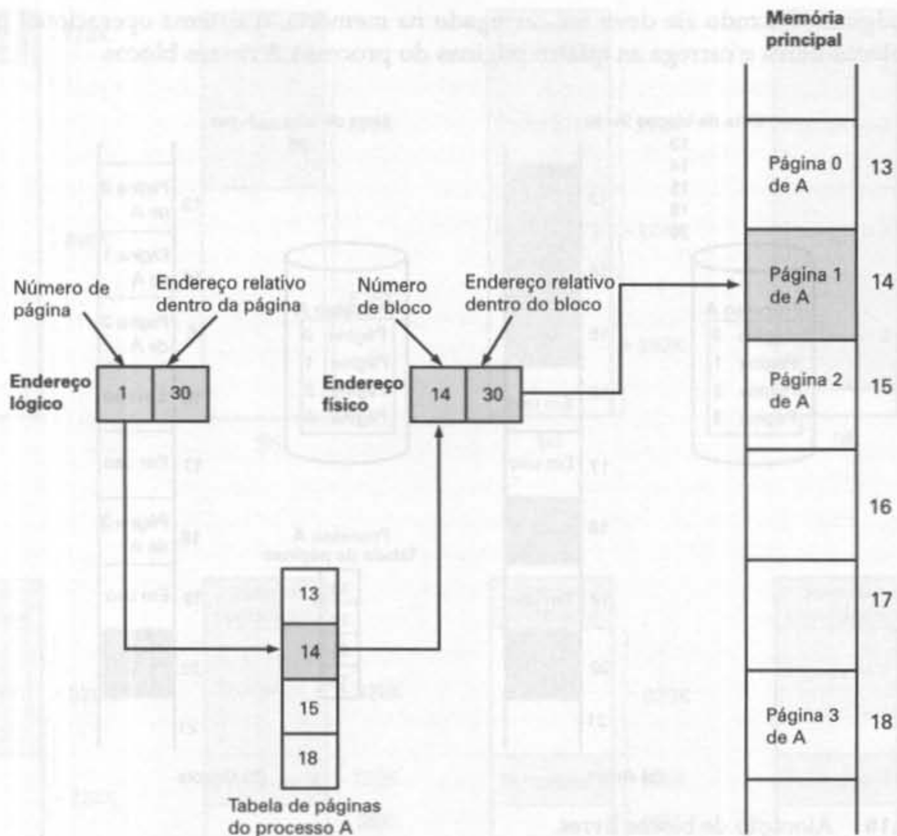


Figura 7.17 Endereços físicos e lógicos.

Memória virtual

Paginação sob demanda

Com o uso do mecanismo de paginação, foi possível desenvolver sistemas com multiprogramação realmente eficientes. Além disso, a simples tática de dividir um processo em páginas levou também ao desenvolvimento de outro conceito importante: a memória virtual.

Para entender o conceito de memória virtual, precisamos acrescentar um refinamento ao esquema de paginação visto anteriormente. Esse refinamento é a *paginação sob demanda*, que significa simplesmente que cada página de um processo é trazida para a memória apenas quando é necessária (isto é, sob demanda).

Considere um processo grande, constituído de um extenso programa e um grande conjunto de dados. De acordo com o princípio de localidade, introduzido no Apêndice 4A, em qualquer curto período de tempo, a execução do processo fica confinada apenas a uma pequena seção do programa (por exemplo, uma sub-rotina) e usa, provavelmente, apenas um ou dois grupos de dados. Claramente, seria um desperdício de memória carregar dúzias de páginas do processo, quando apenas poucas páginas seriam usadas antes que o processo fosse suspenso. Portanto, a memória pode ser bem utilizada, apenas se algumas páginas de cada

processo são carregadas. Nesse caso, se um programa desvia para uma instrução localizada em uma página que não está na memória ou se os dados referenciados não estão na memória, ocorre uma interrupção de *falta de página*. Isso indica ao sistema operacional que ele deve carregar a página desejada.

Assim, existe na memória, a cada instante, apenas um pequeno conjunto das páginas de cada processo, o que possibilita manter maior número de processos na memória. Além disso, economiza-se tempo, pois as páginas de um processo que não foram usadas não precisam ser carregadas nem removidas da memória. No entanto, o sistema operacional precisa saber como gerenciar esse esquema. Ao trazer uma página para a memória, outra página deve ser retirada. Se uma página for retirada um pouco antes de ser usada, ela terá de ser trazida de volta para a memória quase imediatamente. Se isso acontecer com freqüência, diremos que ocorre *thrashing*: o processador gasta a maior parte do tempo com trocas de páginas, em vez de executar instruções. A busca de mecanismos para evitar essas trocas freqüentes foi uma das áreas de pesquisa mais ativas nos anos 70, o que resultou em uma variedade de algoritmos complexos, porém efetivos. Essencialmente, o sistema operacional procura adivinhar, com base na história recente da execução do processo, quais páginas possuem menor probabilidade de serem usadas em um futuro próximo.

Com a paginação sob demanda, não é mais necessário carregar um processo inteiro na memória principal. Isso tem uma conseqüência notável: *é possível que um processo seja maior que toda a área da memória principal*. Uma das restrições mais fundamentais na programação foi eliminada. Sem a paginação sob demanda, o programador tem de saber o tamanho total da memória disponível e, caso seu programa seja muito grande, ele deve imaginar maneiras de estruturá-lo em partes que possam ser carregadas uma de cada vez. Com a paginação sob demanda, essa tarefa é deixada a cargo do sistema operacional e do hardware. O programador pode usar uma memória imensa, de tamanho comparável ao da área disponível em disco.

Como um processo é executado apenas quando carregado na memória principal, essa memória é chamada de *memória real* ou *física*. Entretanto, um programador ou usuário vê uma área de memória muito maior correspondente à memória disponível em disco. Por isso, essa última é conhecida como *memória virtual*. O uso de memória virtual aumenta a eficiência da multiprogramação e evita que o tamanho de programas seja limitado pelo tamanho da memória principal.

Estrutura da tabela de páginas

O mecanismo básico para ler uma palavra na memória envolve a tradução de um endereço virtual, ou lógico, constituído de um número de página e de um endereço relativo na página, para um endereço físico, constituído de um número de bloco e um endereço relativo no bloco, usando uma tabela de páginas. Como a tabela de páginas tem tamanho variável, dependendo do tamanho do processo, não é possível mantê-la em registradores. Para que possa ser usada, ela tem de estar carregada na memória principal. A Figura 7.17 sugere uma implementação desse esquema em hardware. Um registrador mantém o endereço inicial da tabela de páginas do processo que está executando. O número de página do endereço virtual é usado como índice nessa tabela, para obter o número do bloco correspondente. Esse número é combinado com o endereço relativo, contido no endereço virtual, para produzir o endereço físico desejado.

Na maioria dos sistemas, existe uma tabela de páginas para cada processo. Cada processo pode ocupar uma grande área de memória virtual. Por exemplo, na arquitetura VAX,

cada processo pode ter até 2^{31} = 2 Gbytes de memória virtual. Usando páginas de 2^9 = 512 bytes, a tabela de páginas de *cada processo* pode ter até 2^{22} entradas. Claramente, a quantidade de memória requerida para tabelas de páginas seria inaceitável. Para superar esse problema, na maioria dos sistemas de memória virtual, as tabelas de páginas são também endereçadas na memória virtual, e não na memória real. Isso significa que as tabelas de páginas também estão sujeitas à paginação. Quando um processo está em execução, parte da sua tabela de páginas tem de estar na memória principal, incluindo a entrada da tabela que corresponde à página que está sendo executada. Alguns processadores usam um esquema de dois níveis para organizar tabelas de páginas muito grandes. Nesse esquema, existe um diretório de páginas, no qual cada entrada aponta para uma tabela de páginas. Dessa maneira, se o tamanho do diretório de páginas é X e o tamanho máximo de uma tabela de páginas é Y, o processo pode ter até $X \times Y$ páginas. Tipicamente, o tamanho máximo de uma tabela de páginas é restrito ao tamanho de uma página. Um exemplo dessa abordagem de dois níveis será visto neste capítulo, na seção que aborda o processador Pentium II.

Uma abordagem alternativa para as tabelas de páginas de um ou dois níveis é o uso de uma tabela de páginas invertida (Figura 7.18). Essa abordagem é usada no AS/400 da IBM, assim como em todos os produtos RISC da IBM, incluindo o PowerPC.

Nessa abordagem, o número de página do endereço virtual é mapeado em uma tabela *hash*, usando uma função de *hashing* simples[1]. A tabela *hash* contém um apontador para a tabela de páginas invertida, que contém as entradas da tabela de páginas. Com essa estrutura, existe uma entrada na tabela *hash* e uma na tabela de páginas invertida para cada página de memória real (bloco), em vez de uma entrada por página virtual. Dessa maneira, a quantidade de memória requerida para tabelas de páginas é uma fração fixa da memória principal, independentemente do número de processos ou da quantidade de páginas da memória virtual. Como mais de um endereço virtual pode ser mapeado em uma mesma entrada da tabela *hash*, uma lista de endereços é associada a cada entrada da tabela. A técnica de *hashing* resulta em listas tipicamente curtas, com um ou dois endereços.

Cache de tradução de endereços

Em princípio, toda referência à memória virtual pode requerer dois acessos à memória física: um para buscar a entrada apropriada na tabela de páginas e outro para buscar os dados desejados. Dessa maneira, o esquema de memória virtual teria o efeito de dobrar o tempo de acesso à memória. Para evitar esse problema, a maioria dos esquemas de memória virtual utiliza uma memória cache especial para entradas da tabela de páginas, conhecida como TLB ou cache de tradução de endereços*. Essa cache funciona do mesmo modo que uma memória cache e contém as entradas da tabela de páginas usadas mais recentemente. A Figura 7.19 contém

---

1. Uma função de *hashing* mapeia um intervalo de números de 0 a M sobre um intervalo de números de 0 a N, em que $M > N$. O resultado da função de *hashing* é usado como índice na tabela *hash*. Como mais de um número pode ser mapeado em um mesmo valor, é possível que um dado número seja mapeado sobre uma entrada da tabela *hash* já ocupada. Nesse caso, o novo item deve *transbordar* (*overflow*) para uma outra posição da tabela *hash*. Tipicamente, o novo item é colocado no primeiro espaço vazio consecutivo, sendo usado um apontador na posição original, para encadear os itens mapeados sobre essa entrada. Para uma discussão mais detalhada sobre tabelas *hash*, veja Stallings (1998).

* N.R.T.: A memória cache usada para entradas da tabela de páginas é usualmente denominada *translation lookaside buffer* (TLB). A TLB também é conhecida como *address translation cache* (cache de tradução de endereços).

um fluxograma que mostra o uso da TLB. Em virtude do princípio de localidade, boa parte das referências à memória virtual será para endereços localizados nas páginas usadas mais recentemente. Por isso, envolve entradas da tabela de páginas que estão armazenadas na cache. Estudos relativos à TLB do VAX demonstram que o uso desse esquema pode melhorar significativamente o desempenho (Clark e Emer, 1985; Satyanarayanan e Bhandarkar, 1981).

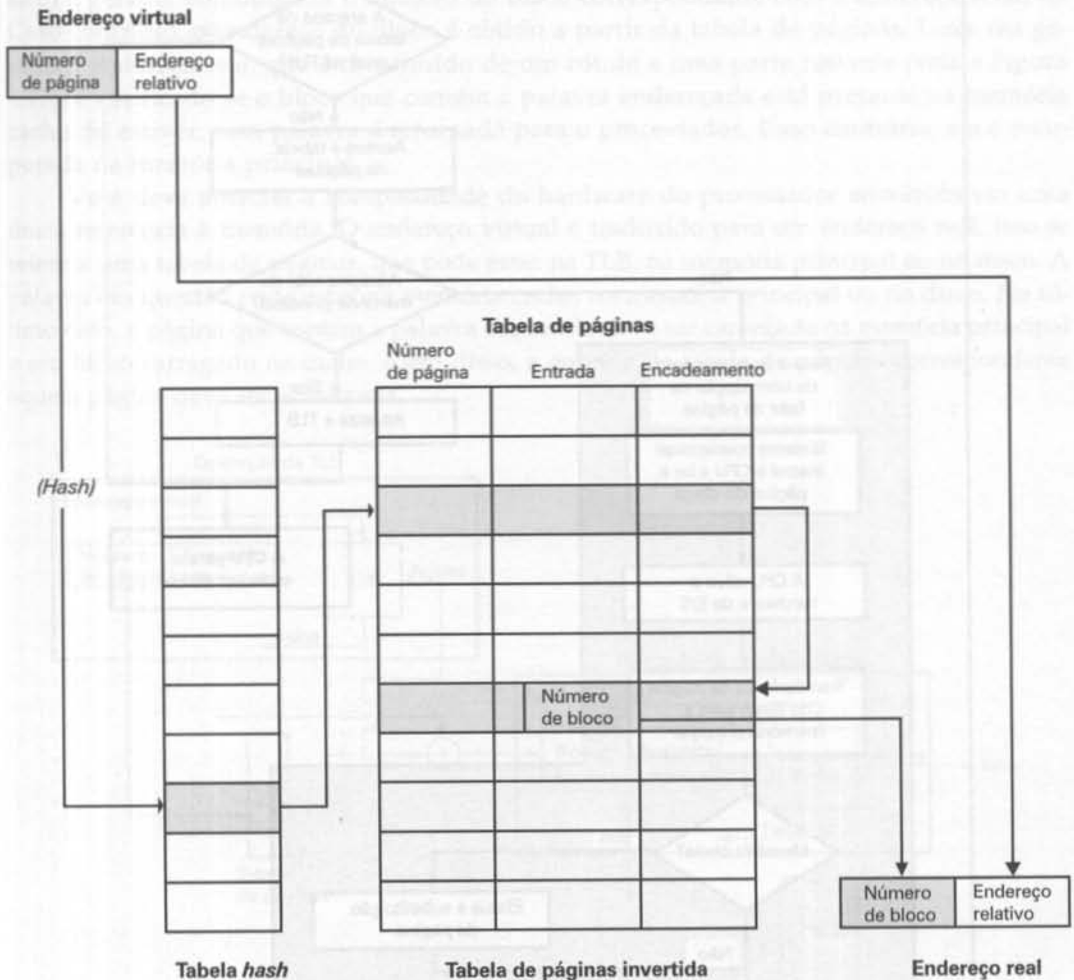


Figura 7.18 Estrutura da tabela de páginas invertida.

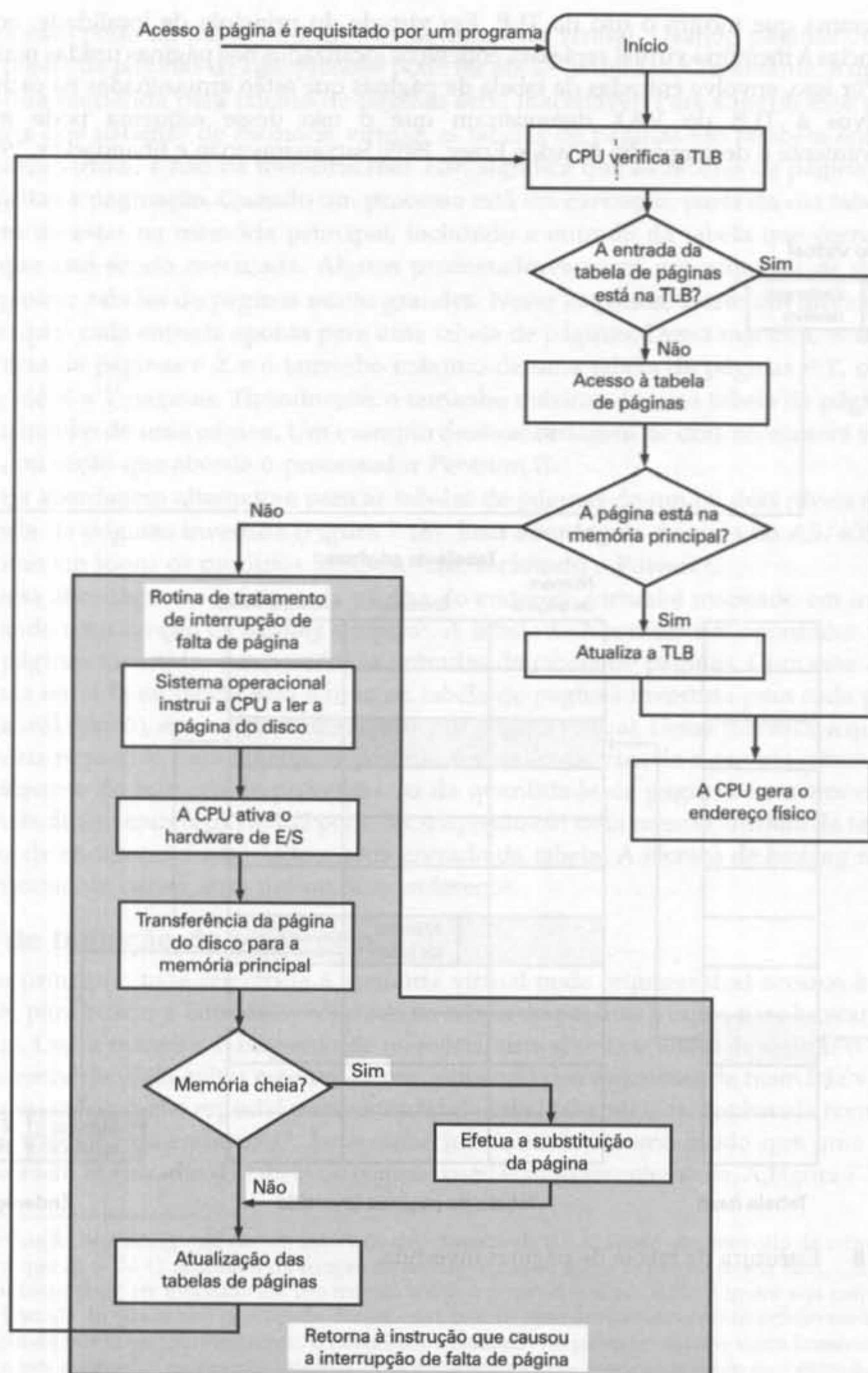


Figura 7.19 Operação do mecanismo de paginação com TLB (Furht e Milutinovic, 1987).

Note que o mecanismo de memória virtual deve interagir com o sistema de memórias cache (não apenas com a memória cache de tradução de endereços mas também com a memória cache da memória principal). Isso é mostrado na Figura 7.20. Um endereço virtual é composto, normalmente, por um número de página e um endereço relativo na página. Primeiramente, o sistema de memória consulta a TLB para verificar se a entrada correspondente à página requerida está presente na cache. Se estiver, o endereço real (físico) é gerado, combinando o número do bloco correspondente com o endereço relativo. Caso contrário, o endereço do bloco é obtido a partir da tabela de páginas. Uma vez gerado o endereço real, que é constituído de um rótulo e uma parte restante (veja a Figura 4.17), é verificado se o bloco que contém a palavra endereçada está presente na memória cache. Se estiver, essa palavra é retornada para o processador. Caso contrário, ela é recuperada da memória principal.

Você deve apreciar a complexidade do hardware do processador envolvido em uma única referência à memória. O endereço virtual é traduzido para um endereço real. Isso se refere a uma tabela de páginas, que pode estar na TLB, na memória principal ou no disco. A palavra em questão pode estar na memória cache, na memória principal ou no disco. No último caso, a página que contém a palavra requerida deve ser carregada na memória principal e seu bloco carregado na cache. Além disso, a entrada da tabela de páginas correspondente àquela página deve ser atualizada.

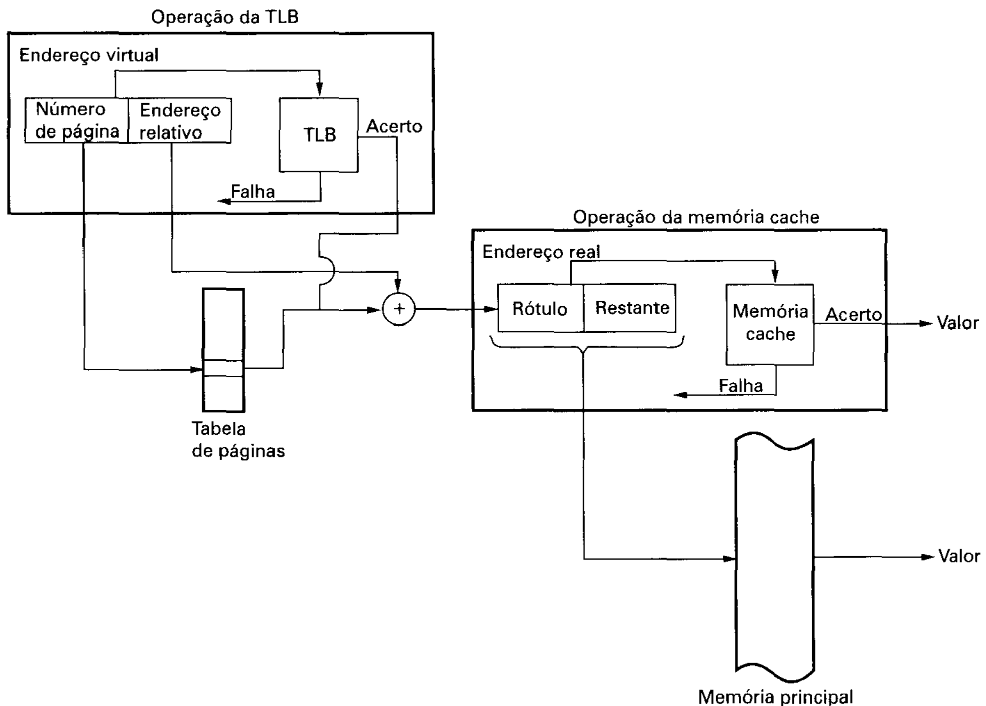


Figura 7.20 Operação da TLB e da memória cache.

Segmentação de memória

Existe ainda uma outra maneira de subdividir o espaço de memória endereçável, conhecida como *segmentação*. Enquanto a paginação é invisível ao programador e visa oferecer um espaço de endereçamento maior, a segmentação normalmente é visível ao programador, sendo usada como um meio conveniente para organizar programas e dados e como um mecanismo para associar atributos de privilégio e de proteção a instruções e dados.

A segmentação possibilita ao programador ver a memória como um conjunto de espaços de endereçamento ou segmentos. O tamanho de cada segmento pode variar dinamicamente. Tipicamente, o programador ou o sistema operacional aloca programas e dados em diferentes segmentos. Podem existir diversos segmentos de programas, de diferentes tipos, assim como diversos segmentos de dados. A cada segmento podem ser atribuídas permissões de acesso e de uso distintas. Uma referência à memória consiste em um número de segmento e um endereço relativo no segmento.

Essa organização apresenta diversas vantagens em relação a um espaço de endereçamento não-segmentado:

1. Simplifica a manipulação de estruturas de dados de tamanho variável. Se o programador não souber *a priori* o tamanho de uma determinada estrutura de dados, ele não precisará adivinhar. Essa estrutura de dados pode ser alocada em um segmento próprio, e o sistema operacional expandirá ou diminuirá o tamanho do segmento quando necessário.
2. Permite que programas possam ser alterados e recompilados independentemente, sem requerer a religação e a recarga de todo o conjunto de programas. Isso é feito usando múltiplos segmentos de programa.
3. Favorece o compartilhamento de código e dados entre processos. Um programador pode colocar um programa utilitário ou uma tabela de dados de uso comum em um segmento que pode ser endereçado por outros processos.
4. Facilita a proteção de memória. Como um segmento pode ser construído de modo a conter um conjunto bem definido de programas ou dados, um programador ou um administrador de sistema pode atribuir privilégios de acesso a programas e dados de maneira conveniente.

Essas vantagens não são disponíveis na paginação, que é invisível para o programador. Por outro lado, a paginação oferece uma maneira eficiente de gerenciamento de memória. Para combinar as vantagens de ambos, alguns sistemas são equipados com hardware e software de sistema operacional para oferecer ambos os esquemas.

7.4 GERENCIAMENTO DE MEMÓRIA DO PENTIUM II E DO PowerPC

Hardware de gerenciamento de memória do Pentium II

Desde a introdução da arquitetura de 32 bits, desenvolveram-se esquemas elaborados de gerenciamento de memória para microprocessadores, com base em lições aprendidas com sistemas de médio e grande portes. Em muitos casos, as versões implementadas para microprocessadores eram superiores aos seus antecessores, voltados para sistemas maiores. Como esses esquemas foram desenvolvidos pelos fabricantes de hardware de microprocessador e podem ser empregados com uma variedade de sistemas operacionais, eles tendem a ser de propósito bastante geral. Um exemplo representativo é o esquema usado no Pentium II. O

hardware de gerenciamento de memória do Pentium II é, essencialmente, o mesmo usado nos processadores Intel 80386 e 80486, com alguns refinamentos.

Espaços de endereçamento

O Pentium II inclui hardware para segmentação e paginação. Esses dois mecanismos podem ser desabilitados, permitindo ao usuário escolher entre quatro opções:

- **Memória não-paginada e não-segmentada:** nesse caso, o endereço virtual é igual ao endereço físico. Isso é útil, por exemplo, em aplicações de controle de baixa complexidade e alto desempenho.
- **Memória não-segmentada e paginada:** nesse caso, a memória é vista como um único espaço de endereçamento linear e paginado. A proteção e o gerenciamento de memória são feitos por meio da paginação. Essa opção é usada por alguns sistemas operacionais (por exemplo, Berkeley UNIX).
- **Memória não-paginada e segmentada:** nesse caso, a memória é vista como um conjunto de espaços de endereçamento lógico. A vantagem dessa opção sobre a abordagem de memória paginada é oferecer proteção ao nível de um único byte, se necessário. Além disso, diferentemente da paginação, ela garante que a tabela de tradução de endereços necessária (a tabela de segmentos) está armazenada na própria pastilha enquanto o segmento estiver na memória. Portanto, a memória segmentada e não-paginada resulta em tempos de acesso previsíveis.
- **Memória paginada e segmentada:** a segmentação é usada para definir as partições da memória lógica sujeitas ao controle de acesso e a paginação, para gerenciar a alocação de memória dentro das partições. Sistemas operacionais como o UNIX System V usam essa opção.

Segmentação

Quando a segmentação é usada, cada endereço virtual (chamado endereço lógico na documentação do Pentium II) é constituído de um número de segmento de 16 bits e um endereço relativo de 32 bits. Dois bits do número de segmento são empregados pelo mecanismo de proteção; os 14 bits restantes especificam um segmento. Dessa maneira, com uma memória não-segmentada, a memória virtual do usuário é de 2^{32} = 4 Gbytes. Com uma memória segmentada, o espaço total de memória virtual do usuário é de 2^{46} = 64 terabytes (Tbytes). O espaço de endereçamento físico utiliza um endereço de 32 bits, tendo tamanho máximo de 4 Gbytes.

O total de memória virtual pode ser, na verdade, maior que 64 Tbytes. Isso ocorre porque a interpretação de um endereço virtual pelo processador depende de qual processo está ativo no momento. O espaço de endereçamento virtual é dividido em duas partes. Metade desse espaço (segmentos de 8 K de 4 Gbytes) é global, sendo compartilhada por todos os processos; a outra metade é local e distinta para cada processo.

Duas formas de proteção são associadas a cada segmento: um nível de privilégio e um atributo de acesso. Os quatro níveis de privilégio variam de 0 (mais protegido) a 3 (menos protegido). O privilégio associado a um segmento de dados constitui sua 'classe' e o associado a um segmento de programa, sua 'permissão'. Um programa em execução pode somente acessar segmentos de dados com nível de privilégio maior (menos privilegiado) ou igual (mesmo privilégio) ao seu nível de permissão.

O hardware não determina como esses níveis de privilégio são usados. Isso depende do projeto e da implementação do sistema operacional. A intenção é que o nível de privilégio 1 seja usado para a maior parte do sistema operacional e o nível 0, para a pequena parte do sistema operacional que cuida de gerenciamento de memória, proteção e controle de acesso. Os outros dois níveis seriam para aplicações. Em muitos sistemas, as aplicações têm nível de prioridade 3 e o nível 2 não é utilizado. Bons candidatos para o nível de prioridade 2 seriam subsistemas especializados de aplicações, que precisam ser protegidos porque implementam seus próprios mecanismos de segurança. Alguns exemplos são sistemas de gerenciamento de banco de dados, sistemas de automatização de escritório e ambientes de engenharia de software.

Além de regular o acesso a segmentos de dados, o mecanismo de privilégio limita o uso de certas instruções. Algumas instruções, tais como as usadas para manipular os registradores de gerenciamento de memória, apenas podem ser executadas no nível 0. Instruções de E/S podem ser executadas até certo nível estabelecido pelo sistema operacional; tipicamente, será o nível 1.

O atributo de acesso de um segmento de dados especifica se ele pode ser usado para leitura e escrita ou apenas para leitura. Para os segmentos de programa, ele especifica se ele pode ser usado para leitura e execução ou apenas para leitura.

No esquema de segmentação, o mecanismo de tradução de endereços envolve o mapeamento de um endereço virtual no que é denominado endereço linear (Figura 7.21b). Um endereço virtual é constituído de um endereço relativo de 32 bits e um seletor de segmento de 16 bits (Figura 7.21a). O seletor de segmento é composto pelos seguintes campos:

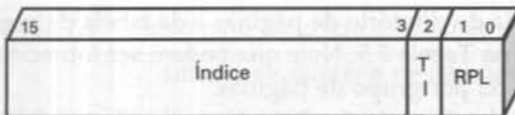
- **Indicador de tabela (*table indicator* — *TI*):** indica se deve ser usada a tabela de segmentos global ou uma tabela de segmentos local para a tradução do endereço.
- **Número de segmento:** número do segmento, que serve como índice na tabela de segmentos.
- **Nível de privilégio requisitado (*requested privilege level* — RPL):** nível de privilégio requerido para esse acesso.

Cada entrada na tabela de segmentos é composta de 64 bits, como mostrado na Figura 7.21c. Os campos de cada entrada são definidos na Tabela 7.5.

Paginação

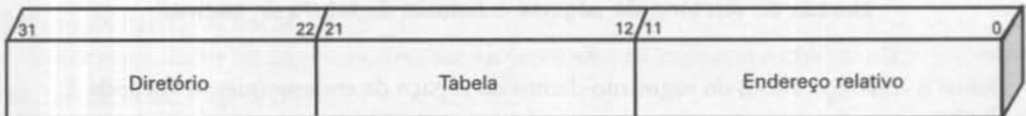
A segmentação é um recurso opcional, que pode ser desabilitada. Quando ela está em uso, os endereços utilizados em programas são os endereços virtuais, que são convertidos em endereços lineares como descrito anteriormente. Quando ela não está em uso, são usados os endereços lineares em programas. Em qualquer um dos casos, o endereço linear tem de ser convertido em um endereço real de 32 bits.

Para entender a estrutura do endereço linear, é preciso estar ciente de que o mecanismo de paginação do Pentium II usa uma tabela de tradução de endereços com dois níveis. O primeiro nível é um diretório de páginas, que contém até 1024 grupos de entradas. Isso divide o espaço da memória linear de 4 Gbytes em grupos de 1024 páginas, cada um com sua própria tabela de páginas e tamanho total de 4 Mbytes. Cada tabela de páginas contém até 1024 entradas; cada entrada corresponde a uma única página de 4 Kbytes. O gerenciamento de memória pode optar por usar um único diretório de páginas para todos os processos, um diretório de páginas para cada processo ou alguma combinação dessas duas opções. O diretório de páginas da tarefa corrente está sempre presente na memória principal. As tabelas de páginas podem estar alocadas na memória virtual.

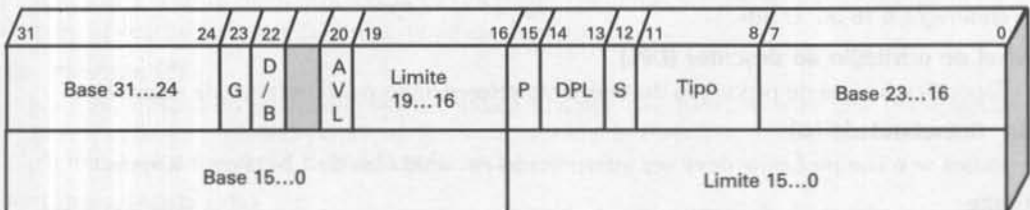


TI = Indicador de tabela
RPL = Nível de privilégio requisitado

(a) Seletor de segmento

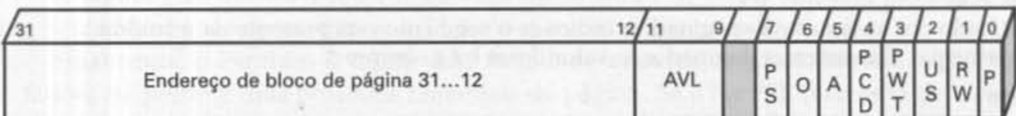


(b) Endereço linear



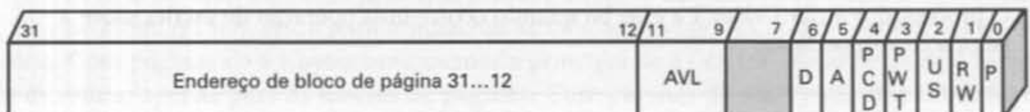
AVL = Disponível para uso pelo software de sistema
Base = Endereço-base do segmento
D/B = Tamanho-padrão de operação
DPL = Nível de privilégio do descritor
G = Granularidade
Limite = Limite de segmento
P = Segmento presente
Tipo = Tipo de segmento
S = Tipo de descritor
= Reservado

(c) Descritor de segmento (entrada da tabela de segmentos)



AVL = Disponível para uso por programadores de sistema
PS = Tamanho da página
A = Bit de acesso
PCD = Desabilitar cache
PWT = Escrita direta
US = Usuário/supervisor
RW = Leitura/escrita
P = Presente

(d) Entrada de diretório de página



D = Modificada

(e) Entrada de tabela de página

Figura 7.21 Formatos para o gerenciamento de memória do Pentium II.

A Figura 7.21 mostra o formato da entrada do diretório de páginas e da tabela de páginas. Os campos de cada entrada são definidos na Tabela 7.5. Note que podem ser fornecidos mecanismos de controle de acesso, por página ou por grupo de páginas.

O Pentium II também usa uma TLB. A cache de tradução de endereços pode manter 32 entradas de tabela de páginas. Quando o diretório de páginas é modificado, a TLB é apagada.

Tabela 7.5 Parâmetros do gerenciamento de memória do Pentium II

Entrada de diretório de páginas e entrada da tabela de páginas

Base

Define o endereço inicial do segmento dentro do espaço de endereçamento linear de 4 Gbytes.

Bit D/B

Quando usado em segmentos de código, indica se o tamanho dos operandos e dos endereços é 16 ou 32 bits.

Nível de privilégio do descritor (DPL)

Especifica o nível de privilégio do segmento referenciado pelo descritor de segmento.

Bit Granularidade (G)

Indica se o campo *Limite* deve ser interpretado em unidades de 1 byte ou 4 Kbytes.

Limite

Define o tamanho do segmento. O processador pode interpretar o campo Limite de duas maneiras, dependendo do valor do bit de granularidade: em unidades de 1 byte, temos um tamanho-limite de segmento de até 1 Mbyte ou em unidades de 4 Kbytes, até um tamanho-limite de segmento de 4 Gbytes.

Bit S

Determina se o segmento é um segmento de sistema ou é um segmento de código ou dados.

Bit Segmento Presente (P)

Usado por sistemas não-paginados. Indica se o segmento está presente na memória principal. Em sistemas paginados, o valor desse bit é sempre 1.

Tipo

Distingue vários tipos de segmento e indica atributos de acesso.

Entrada de diretório de páginas e entrada da tabela de páginas

Bit de acesso (A)

Quando é feita uma operação de leitura ou de escrita sobre uma página, o processador atribui valor 1 ao bit de acesso da entrada correspondente à página, tanto no diretório de páginas quanto na tabela de páginas.

Bit Página Modificada (D)

O processador atribui valor 1 a esse bit quando ocorre uma operação de escrita sobre a página correspondente.

Tabela 7.5 Parâmetros do gerenciamento de memória do Pentium II (*continuação*)

Entrada de diretório de páginas e entrada da tabela de páginas
Endereço de bloco de página
Na tabela de páginas, fornece o endereço físico da página na memória, se o bit P tiver valor 1. Como os blocos são alinhados em limites de 4 K, os 12 bits menos significativos têm valor 0 e, portanto, apenas os 20 bits mais significativos são incluídos na entrada. No diretório de páginas, fornece o endereço de uma tabela de páginas.
Bit Desabilitação da Cache (PCD)
Indica se os dados da página podem ser armazenados na memória cache ou não.
Bit Tamanho de Página (PS)
Indica se o tamanho da página é de 4 Kbytes ou 4 Mbytes.
Bit Escrita Direta da Página (PWT)
Indica se a política de escrita usada para a página pelo mecanismo de armazenamento na memória cache é de escrita direta ou de escrita de volta.
Bit Presente (P)
Em uma entrada da tabela de páginas, indica se a página correspondente está presente na correspondente memória principal. Em uma entrada de diretório, indica se a tabela de páginas está presente na memória principal.
Bit Leitura/Escrita (RW)
Usado para páginas com nível de usuário. Indica se a página pode ser usada para leitura e escrita ou apenas para leitura, por um programa de usuário.
Bit Usuário/Supervisor (U/S)
Indica se a página pode ser usada apenas pelo sistema operacional (modo supervisor) ou pelo sistema operacional e pelas aplicações (modo usuário).

A Figura 7.22 mostra a combinação dos mecanismos de segmentação e paginação. Por simplicidade, não são apresentados a TLB e o mecanismo de memórias cache.

Finalmente, o Pentium II inclui uma facilidade adicional, não encontrada no 80386 ou no 80486: ele permite dois possíveis tamanhos de página. Se o bit PSE (extensão de tamanho de página) do registrador de controle 4 tem valor 1, a unidade de paginação permite ao programador do sistema operacional definir se o tamanho da página é de 4 Kbytes ou 4 Mbytes.

Quando são usadas páginas de 4 Mbytes, o mecanismo de tradução de endereços tem apenas um nível de consulta a tabelas de páginas. Quando o hardware consulta o diretório de páginas, o bit PS da entrada correspondente no diretório de páginas (Figura 7.21d) tem valor 1. Nesse caso, os bits 9 a 21 são ignorados e os bits 22 a 31 definem o endereço inicial de uma página de 4 Mbytes na memória. Dessa maneira, existe uma única tabela de página.

No caso de uma memória principal muito grande, o uso de páginas de 4 Mbytes reduz a área de memória requerida para armazenar as tabelas envolvidas no gerenciamento de memória. Com páginas de 4 Kbytes uma memória principal de 4 Gbytes requer cerca de 4 Mbytes de memória, apenas para as tabelas de páginas. Com páginas de 4 Mbytes, uma única tabela de apenas 4 Kbytes é suficiente para o gerenciamento de memória.

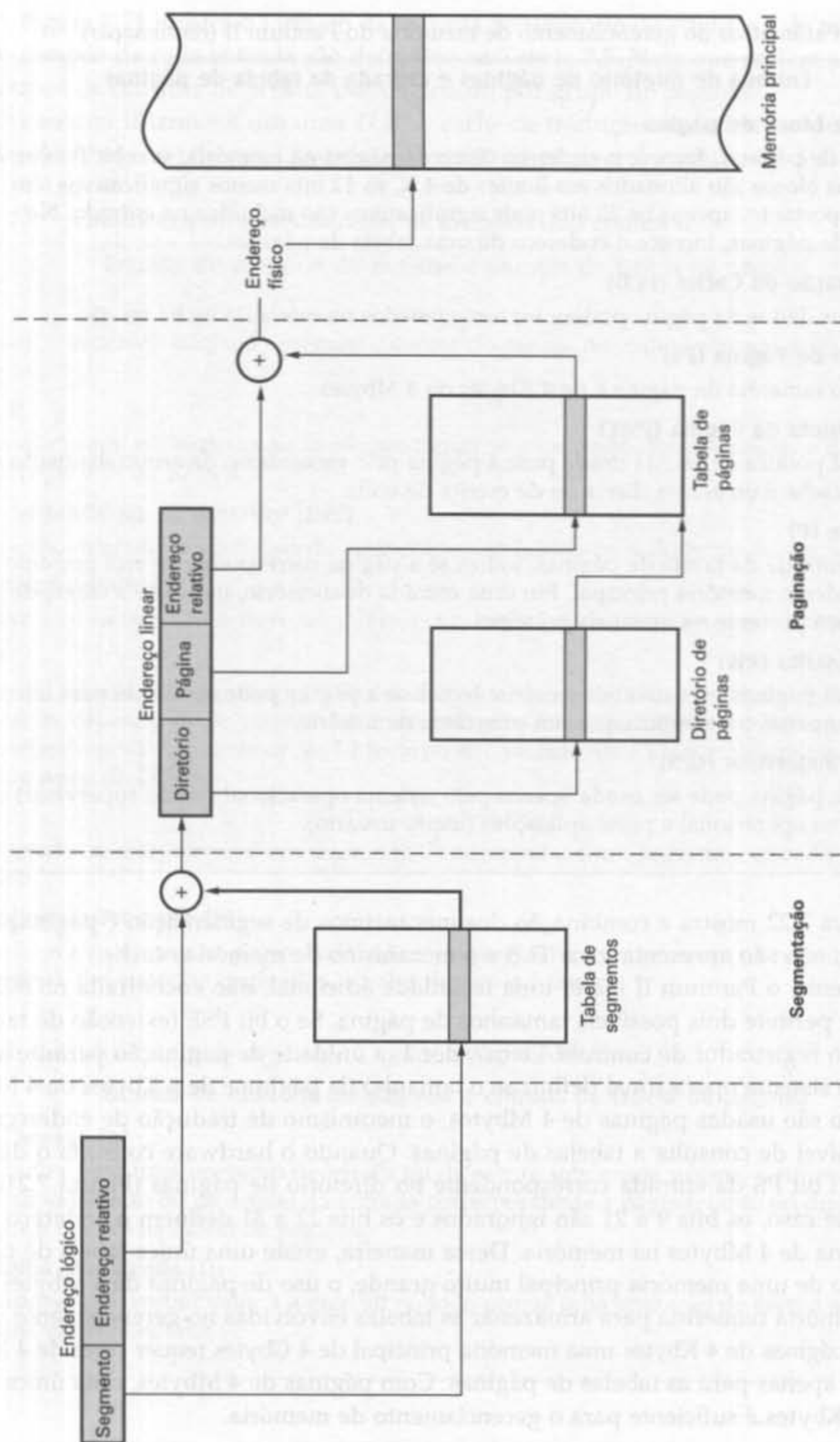


Figura 7.22 Mecanismo de tradução de endereços de memória do Pentium II.

Hardware de gerenciamento de memória do PowerPC

O PowerPC contém um amplo conjunto de mecanismos de endereçamento. Nas implementações da arquitetura para 32 bits, é usado um esquema de paginação com um mecanismo simples de segmentação. Nas implementações com 64 bits, um esquema de paginação com mecanismos mais poderosos de segmentação é utilizado. Além disso, tanto nos processadores de 32 bits quanto nos de 64 bits, existe um mecanismo alternativo de hardware, conhecido como tradução de endereço de bloco. Resumidamente, o esquema de endereçamento de bloco é projetado para resolver uma desvantagem dos mecanismos de paginação. Com a paginação, pode haver grande número de páginas freqüentemente referenciadas por um programa. Por exemplo, programas que usam tabelas do sistema operacional ou áreas de armazenamento temporário de dados gráficos podem exibir esse tipo de comportamento. Como resultado, páginas usadas com freqüência são constantemente movidas para o disco e de volta para a memória. O mecanismo de endereçamento de bloco possibilita ao processador mapear na memória quatro grandes blocos de instruções e quatro grandes blocos de dados, ignorando o mecanismo de paginação.

Uma discussão sobre o mecanismo de endereçamento de blocos está além do escopo deste capítulo. Nesta subseção, concentramo-nos nos mecanismos de paginação e segmentação do PowerPC de 32 bits. O esquema de 64 bits é semelhante.

O PowerPC de 32 bits usa um endereço efetivo de 32 bits (Figura 7.23a). O endereço inclui um seletor de byte de 12 bits e um identificador de página de 16 bits. Dessa maneira, são usadas páginas com tamanho de 2^{12} = 4 Kbytes, sendo permitidas até 2^{16} = 64K páginas por segmento. Quatro bits do endereço são utilizados para designar um dos 16 registradores de segmento. O conteúdo desses registradores é controlado pelo sistema operacional. Cada registrador de segmento inclui bits de controle de acesso, além de um identificador de 24 bits. Assim, o endereço efetivo de 32 bits é mapeado em um endereço virtual de 52 bits (Figura 7.24).

O PowerPC usa uma única tabela de páginas invertida. O endereço virtual é utilizado como índice, nessa tabela, da seguinte maneira. Primeiramente, é computado um código de *hashing* do seguinte modo:

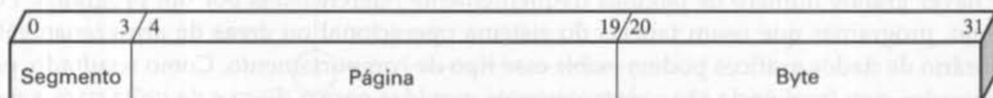
$$H(0...18) = SID(5...23) \oplus VPN(0...18)$$

O número de página do endereço virtual é acrescido de três bits mais significativos, com valor zero, de modo que forme um número de 19 bits (VPN). Então, é calculada uma operação de ou-exclusivo bit a bit desse número com os 19 bits mais à direita do identificador de segmento virtual (SID), para formar um código de *hashing* de 19 bits. A tabela é organizada em *n* grupos de oito entradas. São usados 10 a 19 bits do código de *hashing* (dependendo do tamanho da tabela de páginas) para selecionar um dos grupos da tabela. O hardware de gerenciamento de memória então testa as oito entradas do grupo selecionado, para verificar se alguma corresponde ao endereço virtual dado.

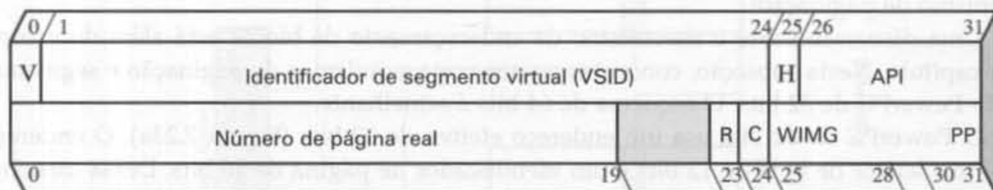
Para fazer esse teste, cada entrada da tabela de páginas inclui um identificador de segmento virtual (VSID), além dos 6 bits mais à esquerda do número da página virtual, denominado índice de página abreviado (porque pelo menos 10 entre os 16 bits do número de página virtual são usados para calcular o código de *hashing*, que seleciona um grupo de entradas da tabela de páginas, apenas uma forma abreviada do número de página virtual precisa ser armazenada na entrada da tabela de páginas). Se alguma entrada na tabela corresponde ao endereço virtual requerido, então um número de página real com 20 bits é concatenado com os

12 bits menos significativos do endereço efetivo, para formar o endereço físico de 32 bits a ser usado.

Se nenhuma entrada no grupo corresponde ao endereço virtual requerido, é calculado um complemento do código de *hashing*, para produzir um novo índice na tabela de páginas, que corresponde à mesma posição relativa do índice anterior, exceto que em relação à extremidade oposta da tabela. O endereço virtual dado é então pesquisado no grupo de páginas selecionado usando esse novo índice. Se o endereço não for encontrado, é gerada uma interrupção de falta de página.

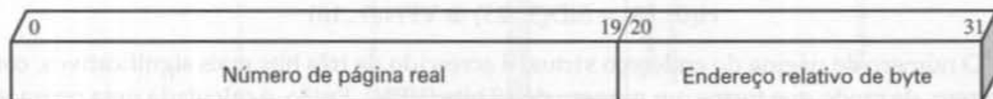


(a) Endereço efetivo



V = Bit Entrada Válida
H = Identificador de função de *hashing*
API = Índice de página abreviado
R = Bit Página Referenciada
C = Bit Página Modificada
WIMG = Bits de controle de acesso à cache e à memória
PP = Bits de proteção de página
= Reservado

(b) Entrada da tabela de páginas



(c) Endereço real

Figura 7.23 Formatos para o gerenciamento de memória do PowerPC de 32 bits.

A Figura 7.23 mostra os formatos do endereço efetivo, da entrada na tabela de páginas e do endereço real e a Figura 7.24, a lógica do mecanismo de tradução de endereços. Finalmente, a Tabela 7.6 define os parâmetros na entrada da tabela de páginas.

O esquema de gerenciamento de memória da implementação de 64 bits é projetado de modo a ser compatível com o da implementação de 32 bits. Essencialmente, todos os endereços efetivos, registradores de uso geral e registradores de endereço de instruções são estendidos de 32 bits para 64 bits, inserindo-se zeros nos bits mais significativos.

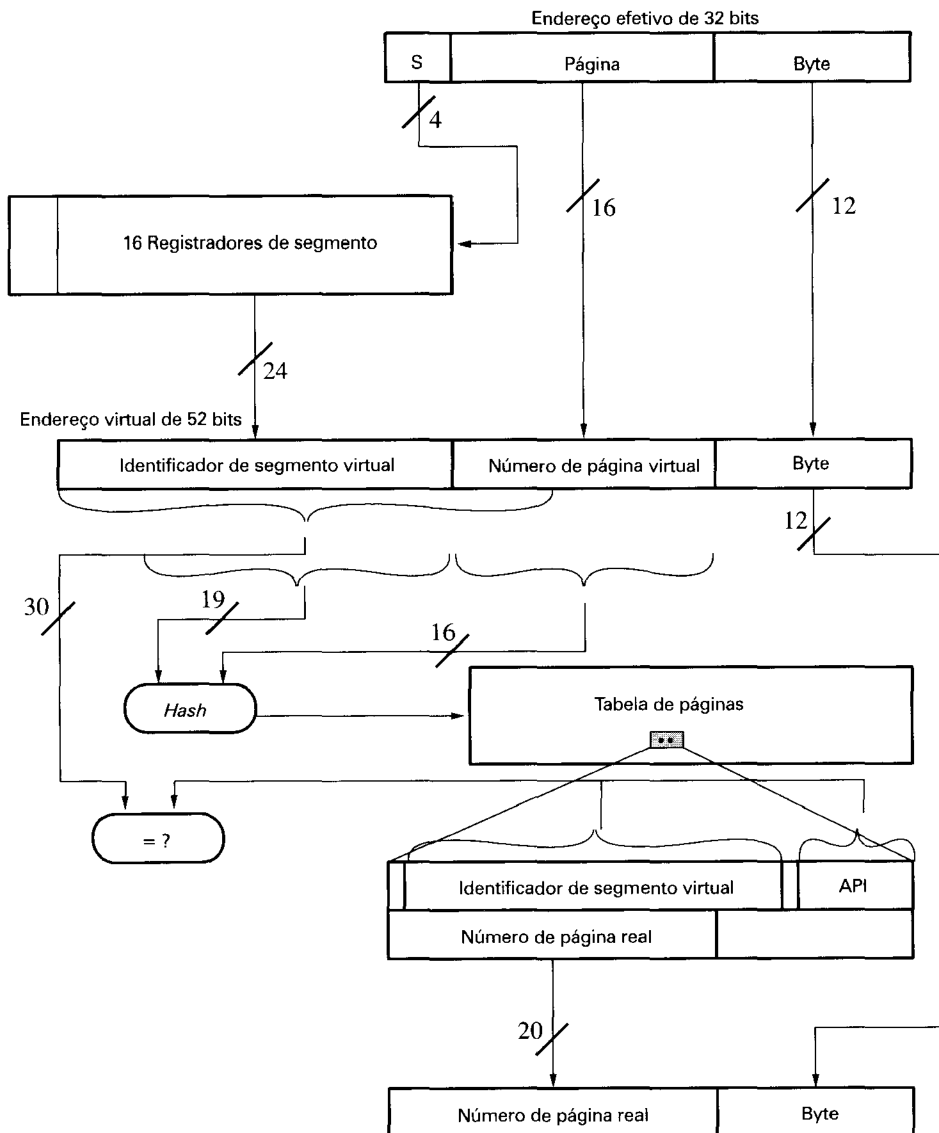


Figura 7.24 Tradução de endereços no PowerPC de 32 bits.

Tabela 7.6 Parâmetros do gerenciamento de memória do PowerPC

Entrada da tabela de segmentos

Identificador de segmento efetivo
Indica um dos 64 G segmentos efetivos; usado para determinar uma entrada na tabela de segmentos.

Bit Entrada Válida (V)
Indica se a entrada possui dados válidos ou não.

Bit Tipo de Segmento (T)
Indica se é um segmento de memória ou de E/S.

Chave de supervisor (Ks)
Usado juntamente com o número de página virtual para determinar uma entrada na tabela de páginas.

Entrada da tabela de páginas

Bit Entrada Válida (V)
Indica se a entrada possui dados válidos ou não.

Identificador de função de hashing (H)
Indica se é uma entrada de *hashing* primária ou secundária.

Índice de página abreviado (API)
Usado para a busca de um endereço virtual.

Bit Página Referenciada (R)
O processador atribui valor 1 a esse bit quando ocorre uma operação de leitura ou escrita na página correspondente.

Bit Página Modificada (C)
O processador atribui valor 1 a esse bit quando ocorre uma operação de escrita na página correspondente.

Bits WIMG
W = 0: usa a política de escrita de volta; W = 1: usa a política de escrita direta.
I = 0: a página pode ser carregada na cache; I = 1: a página não pode ser carregada na cache.
M = 0: página não-compartilhada; M = 1: página compartilhada.
G = 0: página não-protegida; G = 1: página protegida.

Bits Proteção de Página (PP)
Bits de controle de acesso usados juntamente com bits de chave (Ks ou Kp) do registrador de segmento ou da entrada da tabela de segmentos, para definir direitos de acesso.

7.5 LEITURA E SITES WEB RECOMENDADOS

Os tópicos deste capítulo são abordados detalhadamente em Stallings (1998)*.



Sites Web recomendados:

- **Open Group Research Institute — Operating System Program:** o Open Group (uma fusão da Open Software Foundation e da X/Open Company) efetua intensa pesquisa e desenvolvimento em diversas áreas de sistemas operacionais.
- **ACM Special Interest Group on Operating Systems:** contém informações sobre conferências e publicações do SIGOPS (grupo de interesse especial em sistemas operacionais).

7.6 EXERCÍCIOS

7.1 Considere um computador multiprogramado, no qual todas as tarefas têm características idênticas. Durante um intervalo de computação T, uma tarefa gasta a metade do tempo em E/S e a outra metade em atividade do processador. Cada tarefa executa um total de N períodos. Suponha que seja usada uma fila de prioridade circular e que as operações de E/S possam se sobrepor com a operação do processador. Defina as seguintes quantidades:

- Tempo de resposta = tempo real para completar uma tarefa
- Taxa de execução de tarefas = número médio de tarefas completadas por período de tempo T
- Utilização do processador = porcentagem de tempo que o processador está ativo (não-ocioso)

Calcule cada uma dessas quantidades para os casos em que há uma, duas e quatro tarefas simultâneas, supondo que o período T seja distribuído de cada uma das seguintes maneiras:

a. A primeira metade para E/S (I/O-bound) e a segunda metade para o processador.

b. O primeiro e quarto quartos para E/S e o segundo e terceiro quartos para o processador.

7.2 Um programa limitado por E/S é um programa que, quando executado sozinho, gasta mais tempo esperando por E/S do que usando o processador. Um programa limitado pelo processador é o oposto. Suponha que um algoritmo de escalonamento de curto prazo favoreça programas que tenham usado pouco tempo do processador no passado recente. Explique por que esse algoritmo favorece os programas limitados por E/S e, ainda, evita situações em que os programas limitados por processamento fiquem sem tempo de processador.

7.3 Um programa computa a seguinte soma das linhas de uma matriz A de dimensão 100×100:

$$C_i = \sum_{j=1}^{n} a_{ij}$$

* N.R.T.: Uma edição mais nova do livro é: Stallings, W. *Operating systems, internals and design principles*. 4ª ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Suponha que o computador use paginação sob demanda, com páginas de tamanho igual a mil palavras, e que o total de memória principal alocada para dados seja de cinco blocos de páginas. A taxa de faltas de páginas seria diferente caso a matriz *A* fosse armazenada na memória virtual por linha ou por coluna? Explique.

7.4 Suponha que a tabela de páginas do processo que está sendo executado no processador seja tal como a apresentada a seguir. Todos os endereços nessa tabela são números decimais, a partir de zero, e são endereços de bytes de memória. O tamanho de uma página é 1024 bytes.

Número de página virtual	Bit Entrada Válida	Bit Página Referenciada	Bit Página Modificada	Número de bloco de página
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. Descreva exatamente como um endereço virtual gerado pela CPU é traduzido para um endereço físico na memória principal.

b. A qual endereço físico, se houver, corresponderia cada um dos seguintes endereços virtuais? (Não tente manipular nenhuma falta de página, se houver).

(i) 1052
(ii) 2221
(iii) 5499

7.5 Por que o tamanho de página em um sistema de memória virtual não deve ser nem muito pequeno nem muito grande?

7.6 A seguinte seqüência de números de páginas virtuais é encontrada no curso de uma execução em um computador com memória virtual:

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

Suponha que seja adotada uma política de substituição da página usada menos recentemente (LRU). Faça um gráfico da taxa de acerto de página (fração de referências a páginas em que a página é encontrada na memória principal) em função da capacidade de páginas da memória principal n, para $1 \leq n \leq 8$. Suponha que a memória principal esteja inicialmente vazia.

7.7 No computador VAX, o endereço de uma tabela de páginas de usuário é um endereço virtual no espaço de sistema. Qual é a vantagem de ter tabelas de páginas de usuários na memória virtual, e não na memória principal? Qual é a desvantagem?

7.8 Considere um sistema de computação com segmentação e paginação. Quando um segmento está na memória, algumas palavras são desperdiçadas na última página. Além disso, para um tamanho de segmento s e um tamanho de página p, existem s/p entradas na tabela de páginas. Quanto menor o tamanho da página, menor é o desperdício na última página do segmento, mas maior é a tabela de páginas. Que tamanho de página minimiza a sobrecarga total?

7.9 Um computador possui uma memória cache, uma memória principal e um disco, usado para memória virtual. Se uma palavra referenciada está na memória cache, o tempo de acesso é de 20 ns. Se está na memória principal, mas não na cache, são necessários 60 ns para carregá-la na cache, sendo a referência então iniciada novamente. Se não está na memória principal, são necessários 12 ms para buscar a palavra no disco, seguidos de 60 ns para copiá-la na cache, e então a referência é novamente iniciada. A taxa de acerto na cache é de 0,9 e na memória principal é de 0,6. Qual é o tempo médio, em nanossegundos, necessário para acessar uma palavra referenciada nesse sistema?

7.10 Suponha que uma tarefa seja dividida em quatro segmentos do mesmo tamanho e que o sistema construa uma tabela de descritores de página com oito entradas para cada segmento. O sistema usa, portanto, uma combinação de segmentação e paginação. Suponha, ainda, que o tamanho da página seja de 2 Kbytes.

a. Qual é o tamanho máximo de cada segmento?

b. Qual é o espaço de endereçamento lógico máximo da tarefa?

c. Suponha que um valor localizado no endereço físico 00021ABC seja acessado pela tarefa. Qual é o formato do endereço lógico que a tarefa gera para esse valor? Qual é o espaço de endereçamento físico máximo do sistema?

Fonte: Alexandridis (1993).

7.11 Considere um microprocessador capaz de endereçar até 2^{32} bytes de memória principal física. Ele implementa um espaço de endereçamento lógico segmentado, com tamanho máximo de 2^{31} bytes. Cada instrução contém um endereço completo de duas partes. São usadas unidades de gerenciamento de memória (*memory-management units* — MMUs) externas, cujo esquema de gerenciamento aloca blocos contíguos de memória física para os segmentos. Esses blocos têm tamanho fixo e igual a 2^{22} bytes. O endereço físico inicial de um segmento é sempre divisível por 1024. Mostre, em detalhes, a interconexão do mecanismo externo de mapeamento, que converte endereços lógicos em físicos, usando um número apropriado de MMUs. Mostre também, detalhadamente, a estrutura interna de uma MMU (considerando que cada MMU possui uma memória cache de descritores de segmento com 128 entradas e com mapeamento direto) e como cada MMU é selecionada.

Fonte: Alexandridis (1993).

7.12 Considere um espaço de endereçamento lógico paginado (composto de 32 páginas de 2 Kbytes cada), mapeado em um espaço de memória física de 1 Mbyte.

a. Qual é o formato do endereço lógico do processador?

b. Qual é o número de entradas na tabela de páginas e qual o tamanho de cada entrada (desconsiderando os bits de "permissão de acesso")?

c. Qual seria o efeito sobre a tabela de páginas, se o espaço físico de memória fosse reduzido pela metade?

Fonte: Alexandridis (1993).

PARTE

3

A Unidade Central de Processamento

OBJETIVOS

Até esse ponto, vimos a CPU essencialmente como uma 'caixa-preta' e consideramos sua interação com a memória e a E/S. A Parte III examina a estrutura e o funcionamento interno da CPU. A CPU consiste de uma unidade de controle, dos registradores, da unidade lógica e aritmética, da unidade de execução de instruções e das interconexões entre esses componentes. Esta parte aborda questões relativas à arquitetura da CPU, tais como o projeto do conjunto de instruções e dos tipos de dados. Além disso, trata também de aspectos de organização, tais como *pipelining*.

ROTEIRO

Capítulo 8 Aritmética computacional

O Capítulo 8 examina a funcionalidade da unidade lógica e aritmética (ULA), enfocando a representação de números e as técnicas para implementar as operações aritméticas. Geralmente, os processadores implementam dois tipos de aritmética: de números inteiros ou ponto fixo e de ponto flutuante. Em ambos os casos, abordamos primeiro a representação dos números e, em seguida, discutimos as operações aritméticas. O importante padrão de ponto flutuante IEEE 754 é examinado em detalhes.

Capítulo 9 Conjunto de instruções: características e funções

O complexo tópico do projeto do conjunto de instruções ocupa os Capítulos 9 e 10. O Capítulo 9 focaliza os aspectos funcionais do projeto do conjunto de instruções. São discutidos os tipos de funções que são especificadas pelas instruções da linguagem de máquina de computadores e então são examinados especialmente os tipos de operandos (que especificam os dados a serem operados) e os tipos de operações (que especificam as operações a serem desempenhadas) comumente encontradas em conjuntos de instruções.

Capítulo 10 Conjunto de instruções: modos de endereçamento e formatos

Enquanto o Capítulo 9 trata da semântica dos conjuntos de instruções, o Capítulo 10 aborda a sintaxe dos conjuntos de instruções. Especificamente, o Capítulo 10 examina o modo como são especificados os endereços da memória e o formato geral das instruções de um computador.

Capítulo 11 Estrutura e funcionamento da CPU

O Capítulo 11 descreve o uso de registradores como a memória interna da CPU e, em seguida, revê todo o material abordado até esse ponto, para oferecer uma visão geral da estrutura e do funcionamento da CPU. A organização geral (ULA, unidade de controle, registradores) é revisada. Depois disso, discutimos a organização do conjunto de registradores. O restante do capítulo descreve o funcionamento do processador durante a execução de instruções de máquina. O ciclo de instruções é examinado para mostrar o funcionamento e a relação entre os ciclos de busca, indireção, execução e interrupção. Finalmente, exploramos detalhadamente o uso de *pipelining* para obter melhor desempenho.

Capítulo 12 Computadores com conjunto reduzido de instruções

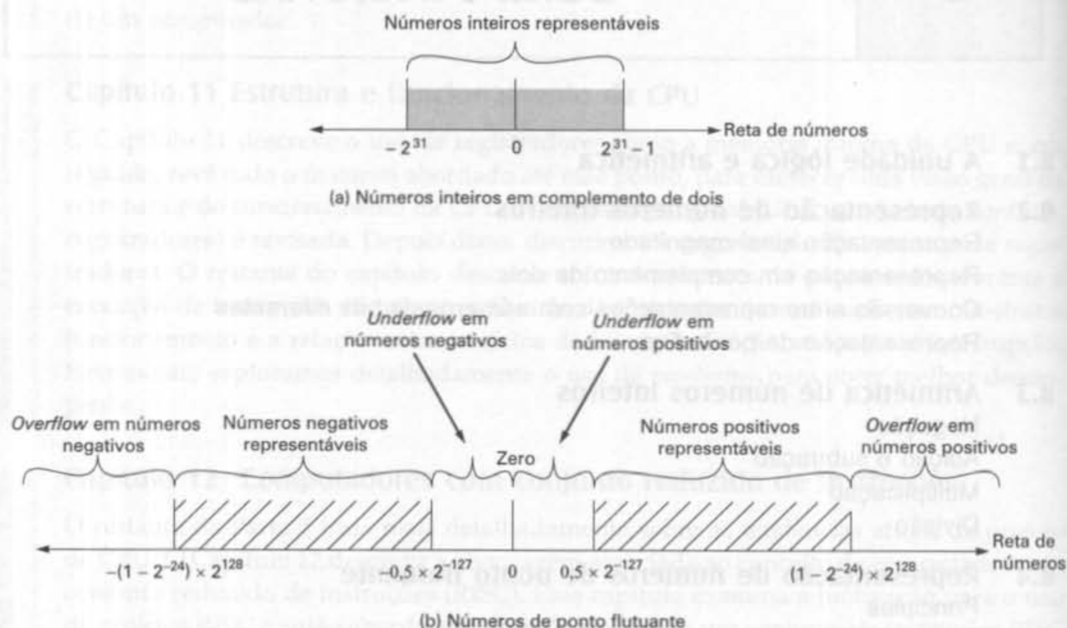
O restante da Parte 3 trata mais detalhadamente sobre as tendências atuais de projeto de CPU. O Capítulo 12 descreve a abordagem associada ao conceito de computador com conjunto reduzido de instruções (RISC). Esse capítulo examina a motivação para o uso de projetos RISC e então aborda detalhes do projeto de um conjunto de instruções RISC e a arquitetura de uma CPU RISC.

Capítulo 13 Paralelismo no nível de instruções e processadores superescalares

O Capítulo 13 trata do uso de técnicas superescalares, abordagem que é usada em muitos projetos de processadores mais recentes.

capítulo

8 ARITMÉTICA COMPUTACIONAL



(a) Números inteiros em complemento de dois

(b) Números de ponto flutuante

- Os dois aspectos mais importantes da aritmética computacional são o modo como os números são representados (o formato binário) e os algoritmos usados para as operações aritméticas básicas (adição, subtração, multiplicação e divisão). Isso se aplica tanto para a aritmética de números inteiros quanto para a de números de ponto flutuante.
- Números de ponto flutuante são expressos na forma de um número (mantissa) multiplicado por uma constante (base) elevada a uma potência inteira (expoente). Eles podem ser usados para representar números muito grandes e muito pequenos.
- A maioria dos processadores implementa o padrão IEEE 754 para representação e aritmética de números de ponto flutuante. Esse padrão define um formato de 32 bits e de 64 bits.

Nosso estudo sobre o processador começa com uma visão geral da unidade lógica e aritmética (ULA — *arithmetic and logic unit*). Em seguida, enfocamos o aspecto mais complexo da ULA: a aritmética computacional. As funções lógicas que fazem parte da ULA são descritas no Capítulo 9 e a implementação de funções lógicas e aritméticas simples por meio de circuitos digitais é discutida no Apêndice A.

A aritmética computacional geralmente opera com dois tipos de números muito diferentes: números inteiros e números de ponto flutuante. Em ambos os casos, a escolha da representação é uma questão crucial de projeto, sendo, por isso, tratada primeiro. As operações aritméticas serão discutidas em seguida.

Uma revisão sobre sistemas de numeração é incluída no apêndice deste capítulo.

8.1 A UNIDADE LÓGICA E ARITMÉTICA

A ULA é a parte do computador que de fato executa as operações aritméticas e lógicas sobre os dados. Todos os outros elementos do computador — unidade de controle, registradores, memória, E/S — servem, principalmente, para trazer os dados a serem processados pela ULA e receber os resultados das operações efetuadas. De certo modo, a ULA constitui o núcleo ou a essência de um computador.

Assim como todos os demais componentes eletrônicos de um computador, a ULA é baseada em dispositivos lógicos digitais simples, capazes de armazenar dígitos binários e efetuar operações simples de lógica booleana. O leitor interessado na implementação de lógica digital pode consultar o apêndice deste livro.

A Figura 8.1 indica, em termos gerais, como a ULA é conectada com o restante do processador. Os dados são fornecidos à ULA em registradores e os resultados de uma operação são armazenados em registradores. Esses registradores são áreas de armazenamento temporário dentro do processador, conectadas à ULA por meio de caminhos de sinal (veja, por exemplo, a Figura 2.3). A ULA pode também ativar bits especiais (*flags*) para indicar o resultado de uma operação. Por exemplo, caso o resultado de uma operação exceda a capacidade de armazenamento de um registrador, isso é indicado atribuindo o valor 1 ao bit de *overflow*. Esses bits especiais são também armazenados em registradores internos do processador. A unidade de controle fornece sinais para controlar a operação da ULA e a transferência de dados entre a ULA e os registradores.

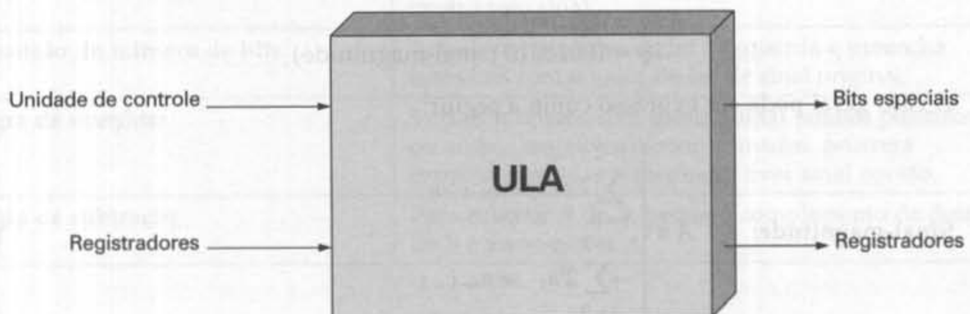


Figura 8.1 Entradas e saídas da ULA.

8.2 REPRESENTAÇÃO DE NÚMEROS INTEIROS

No sistema de números binários (veja o Apêndice 8A), é possível representar números arbitrários usando os dígitos zero e um, o sinal de subtração ("–", para números negativos) e a vírgula decimal (que separa a parte inteira e a parte fracionária do número). Por exemplo:

$$-1101{,}0101_2 = -13{,}3125_{10}$$

Entretanto, para armazenar e processar esses números no computador, não é possível usar os sinais de menos e vírgula. Apenas dígitos binários (0 e 1) podem ser usados para a representação de números. Isso não constitui um problema se quisermos apenas representar números inteiros não negativos. Por exemplo, uma palavra de 8 bits pode ser usada para representar números de 0 a 255.

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

De modo geral, se uma seqüência de n dígitos binários $a_{n-1} a_{n-2} \ldots a_1 a_0$ for interpretada como um número inteiro sem sinal A, seu valor será dado por:

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Representação sinal-magnitude

Diversas convenções alternativas são usadas para representar números inteiros positivos e negativos; todas elas tratam o bit mais significativo da palavra (bit mais à esquerda) como um bit de sinal: se o bit mais à esquerda for 0, o número será positivo; se for 1, o número será negativo.

A forma mais simples de representação que emprega um bit de sinal é a representação sinal-magnitude. Em uma palavra de n bits, os $n - 1$ bits mais à direita representam a magnitude do número inteiro. Por exemplo:

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \text{ (sinal-magnitude)} \end{aligned}$$

O caso geral pode ser expresso como a seguir:

Sinal-magnitude: $$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{se } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{se } a_{n-1} = 1 \end{cases} \qquad (8.1)$$

A representação sinal-magnitude apresenta diversas desvantagens. Uma delas é que, para efetuar operações de adição e subtração, é preciso considerar tanto a magnitude quanto

o sinal dos dois operandos. Isso ficará mais claro na discussão apresentada na Seção 8.3. Outra desvantagem é que existem duas representações para 0:

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \text{ (sinal-magnitude)}$$

Isso é inconveniente, pois é mais difícil testar se um valor é igual a 0 (operação que é executada freqüentemente) do que no caso em que existe uma única representação para 0.

Em virtude dessas desvantagens, a representação sinal-magnitude é raramente usada na implementação da parte inteira de uma ULA. O esquema mais comum é a representação em complemento de dois.

Representação em complemento de dois

Assim como a representação sinal-magnitude, a representação em complemento de dois usa o bit mais significativo como bit de sinal, o que torna fácil testar se um número inteiro é positivo ou negativo. Entretanto, os demais bits são interpretados de maneira diferente. A Tabela 8.1 relaciona as características-chave da representação e da aritmética em complemento de dois, abordadas nesta e na próxima seção.

A maioria das abordagens da representação em complemento de dois define apenas as regras para representar os números negativos, não incluindo qualquer prova formal de que esse esquema 'funciona'. A abordagem adotada nesta seção e na Seção 8.3, ao contrário, é baseada em Dattatreya (1993), que sugere que essa representação será bem compreendida se for definida em termos de uma soma ponderada de bits, como foi feito anteriormente para o caso das representações sem sinal e de sinal-magnitude. A vantagem desse tratamento é não deixar dúvidas de que as regras definidas para as operações aritméticas na notação em complemento de dois funcionam em todos os casos.

Tabela 8.1 Características da representação e aritmética em complemento de dois

Faixa de valores representáveis	-2^{n-1} a $2^{n-1} - 1$
Número de representações para zero	1
Negação	Pegue o complemento booleano de cada bit do número positivo correspondente e então some 1 ao padrão de bits resultante, tratado como um número inteiro sem sinal.
Expansão do número de bits	Acrescente posições de bit à esquerda e preencha esses bits com o valor do bit de sinal original.
Regra de *overflow*	Se dois números com mesmo sinal (ambos positivos ou ambos negativos) forem somados, ocorrerá *overflow* apenas se o resultado tiver sinal oposto.
Regra de subtração	Para subtrair B de A, pegue o complemento de dois de B e some-o com A.

Considere um número inteiro A de n bits na representação em complemento de dois. Se A for positivo, então o bit de sinal, a_{n-1}, será igual a zero. Os bits restantes representam a magnitude do número, assim como na representação sinal-magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \text{ para } A \geq 0$$

O número zero é tratado como um número positivo, isto é, tem o bit de sinal igual a 0 e todos os demais bits (magnitude) iguais a 0. É fácil ver que a faixa de números inteiros positivos que podem ser representados é de 0 (todos os bits de magnitude são iguais a 0) a $2^{n-1} - 1$ (todos os bits de magnitude são iguais a 1). Para representar um número maior seriam necessários mais bits.

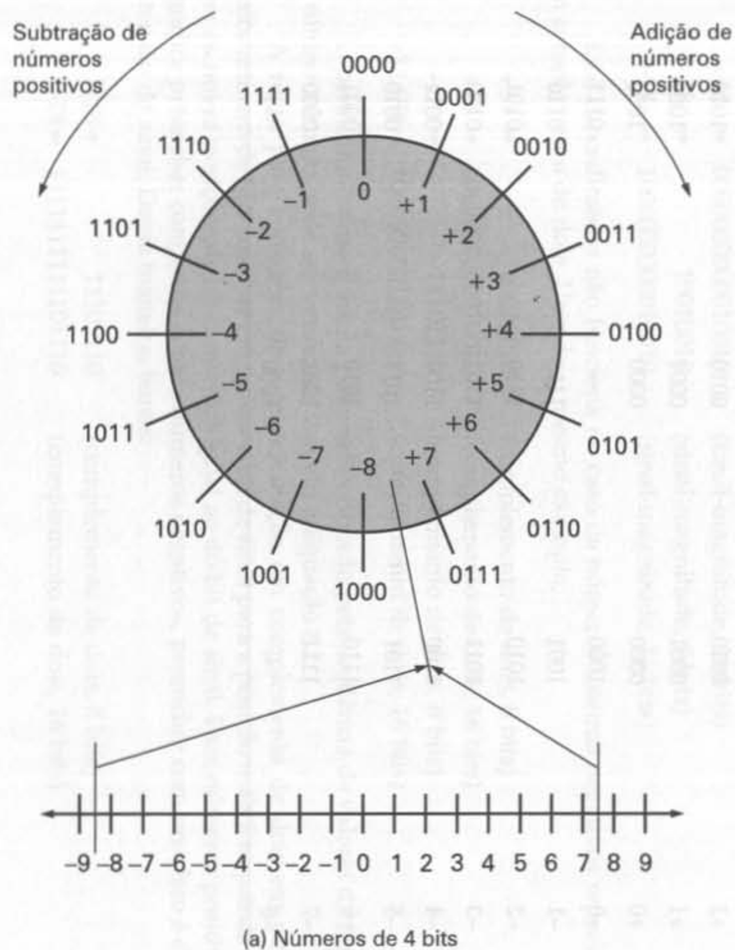
Se A é um número negativo ($A < 0$), o bit de sinal, a_{n-1}, é 1. Os $n - 1$ bits restantes podem representar até 2^{n-1} valores. Portanto, a faixa de números inteiros negativos que podem ser representados é de -1 a -2^{n-1}. É desejável associar padrões de bits a números inteiros negativos, de maneira que as operações aritméticas possam ser efetuadas diretamente, de modo semelhante à aritmética de números inteiros sem sinal. Na representação de números inteiros sem sinal, para calcular o valor de um número inteiro a partir da sua representação, o bit mais significativo é multiplicado por $+2^{n-1}$. Em uma representação que usa um bit de sinal, as propriedades aritméticas desejadas são obtidas, como veremos na Seção 8.3, se o valor do número é calculado a partir da sua representação multiplicando o bit mais significativo por -2^{n-1}. Essa é a convenção adotada na representação em complemento de dois, o que fornece a seguinte expressão para números negativos:

Complemento de dois $$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad (8.2)$$

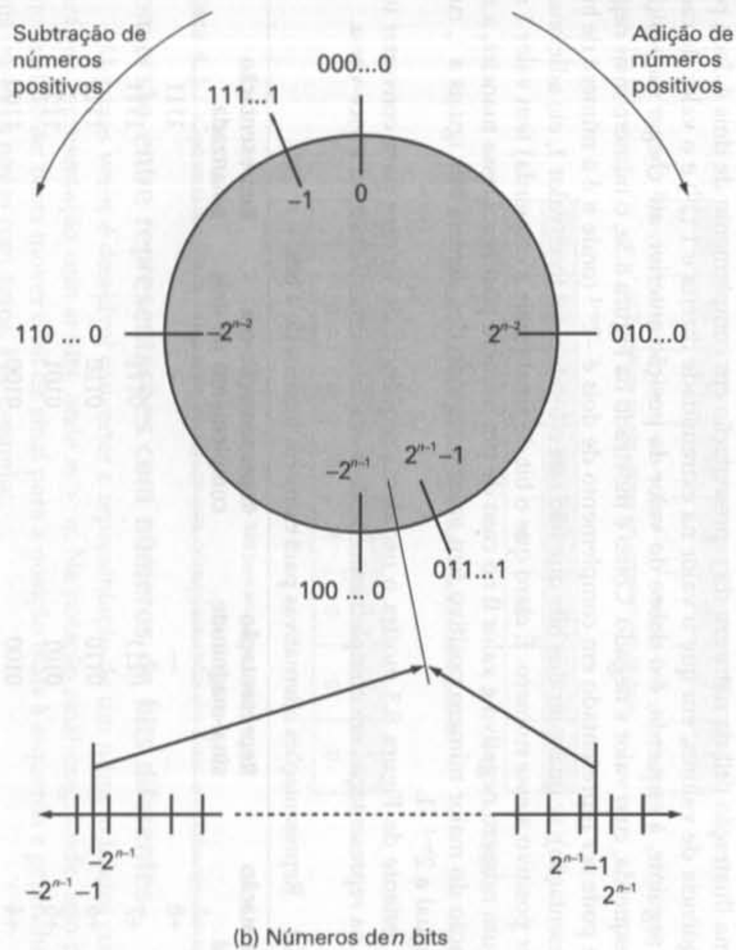
Para números inteiros positivos, $a_{n-1} = 0$. Portanto, a Equação 8.2 define o valor da representação em complemento de dois, tanto para números positivos quanto para negativos.

A representação em complemento de dois pode ser visualizada por meio da representação geométrica mostrada na Figura 8.2, extraída de Benham (1992). O círculo na metade superior de cada parte da figura é formado selecionando o segmento adequado da reta de números e juntando as duas extremidades. Começando a partir de qualquer número do círculo, podemos somar um valor k positivo (ou subtrair um valor k negativo) a esse número, movendo k posições no sentido horário, ou subtrair um valor k positivo (ou somar um valor k negativo), movendo k posições no sentido anti-horário. Caso essa operação cruze o ponto em que as duas extremidades se juntam, a resposta obtida é incorreta.

A Tabela 8.2 compara as representações sinal-magnitude e em complemento de dois para números inteiros de 4 bits. Embora a representação em complemento de dois possa parecer pouco natural, do ponto de vista humano, veremos que ela torna mais fácil a implementação das operações aritméticas mais importantes — a adição e a subtração. Por isso, ela é usada quase universalmente para representar números inteiros no processador.



(a) Números de 4 bits



(b) Números de n bits

Figura 8.2 Representação geométrica dos números inteiros em complemento de dois.

Uma ilustração útil da natureza da representação em complemento de dois é dada por uma seqüência de valores, em que o valor na extremidade direita é 1 (2^0) e o valor de cada posição seguinte, à esquerda, é o dobro do valor da posição anterior, até chegar na posição mais à esquerda, cujo valor é negado. Como é mostrado na Figura 8.3a, o número mais negativo que pode ser representado em complemento de dois é -2^{n-1} (onde n é o número de bits da representação); se qualquer dos bits que não seja o bit de sinal tiver valor 1, ele adicionará um valor positivo a esse número. É claro que o bit de sinal (mais à esquerda) tem valor 1 no caso de um número negativo e valor 0 no caso de um número positivo. Dessa maneira, a representação do maior número positivo tem valor 0 seguido dos demais bits iguais a 1, cujo valor é igual a $2^{n-1}-1$.

O restante da Figura 8.3 mostra o uso dessa seqüência de valores para converter um número na representação em complemento de dois em um número decimal e vice-versa.

Tabela 8.2 Representações alternativas para números inteiros de 4 bits

Representação decimal	Representação sinal-magnitude	Representação em complemento de dois	Representação polarizada
+8	—	—	1111
+7	0111	0111	1111
+6	0110	0110	1110
+5	0101	0101	1101
+4	0100	0100	1100
+3	0011	0011	1011
+2	0010	0010	1010
+1	0001	0001	1001
+0	0000	0000	1000
–0	1000	—	0111
–1	1001	1111	0110
–2	1010	1110	0101
–3	1011	1101	0100
–4	1100	1100	0011
–5	1101	1011	0010
–6	1110	1010	0001
–7	1111	1001	0000
–8	—	1000	—

-128	64	32	16	8	4	2	1

(a) Seqüência de oito valores em complemento de dois

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Conversão do valor binário 10000011 para um valor decimal

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Conversão do valor decimal −120 para seu valor binário

Figura 8.3 Conversão entre números binários em complemento de dois e números decimais.

Conversão entre representações com números de bits diferentes

Algumas vezes é desejável converter a representação de um número inteiro com n bits para sua representação com m bits, onde $m > n$. Na notação sinal-magnitude, isso pode ser feito facilmente: basta mover o bit de sinal para a posição mais à esquerda e preencher as demais posições novas com zeros. Por exemplo:

+18 = 00010010 (sinal-magnitude, 8 bits)
+18 = 0000000000010010 (sinal-magnitude, 16 bits)
−18 = 10010010 (sinal-magnitude, 8 bits)
−18 = 1000000000010010 (sinal-magnitude, 16 bits)

Esse procedimento não funciona no caso de números inteiros negativos representados em complemento de dois. Usando o mesmo exemplo,

+18 = 00010010 (complemento de dois, 8 bits)
+18 = 0000000000010010 (complemento de dois, 16 bits)
−18 = 11101110 (complemento de dois, 8 bits)
−32.658 = 1000000001101110 (complemento de dois, 16 bits)

O penúltimo número acima pode ser verificado pela seqüência de valores da Figura 8.3. O último número pode ser verificado usando a Equação 8.2.

A regra para converter uma representação em complemento de dois em outra com maior número de bits consiste em mover o bit de sinal para a posição mais à esquerda e preencher as novas posições de bit com valor igual ao do bit de sinal. Para números positivos, isso significa preencher com zeros e, para números negativos, preencher com uns. Isso é chamado extensão de sinal. Dessa maneira, temos:

−18 = 11101110 (complemento de dois, 8 bits)
−18 = 1111111111101110 (complemento de dois, 16 bits)

Para mostrar que essa regra funciona, considere uma seqüência de n bits de dígitos binários $a_{n-1}a_{n-2} \dots a_1a_0$, interpretada como um número inteiro A, em complemento de dois, de modo que seu valor seja:

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Se A for um número positivo, a regra claramente funcionará. Se A for negativo e quisermos obter sua representação com m bits, onde $m > n$, então,

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

Esses dois valores devem ser iguais:

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow a_{m-1} = a_{m-2} = \dots = a_{n-2} = a_{n-1} = 1$$

Ao passar da primeira para a segunda equação, pede-se que os $n - 1$ bits menos significativos permaneçam inalterados de uma representação para outra. Obtemos, então, a penúltima equação, que é verdadeira somente se todos os bits da posição $n - 1$ a $m - 2$ são iguais a 1. Dessa maneira, a regra de extensão de sinal funciona.

Representação de ponto fixo

Mencionamos anteriormente que as representações discutidas nesta seção são conhecidas como representações de ponto fixo. Isso ocorre porque elas fixam a posição da vírgula decimal como a posição à direita do bit menos significativo. Essas representações podem também ser usadas pelo programador para frações binárias, se supusermos a vírgula decimal posicionada implicitamente em outra posição.

8.3 ARITMÉTICA DE NÚMEROS INTEIROS

Esta seção examina a implementação das operações aritméticas mais comuns em números representados em complemento de dois.

Negação

Na representação sinal-magnitude, a regra para a negação de um número inteiro é simples: basta inverter o valor do bit de sinal. Na notação em complemento de dois, a negação de um número inteiro é obtida pelos seguintes passos:

1. Tome o complemento booleano de cada bit do número (incluindo o bit de sinal), isto é, troque cada 1 por 0 e cada 0 por 1.
2. Adicione 1 ao resultado, visto como um número inteiro binário sem sinal.

Esses dois passos do processo fornecem a operação de *complemento de dois* de um número inteiro. Por exemplo:

```
                   +18 = 00010010 (complemento de dois)
complemento bit a bit  = 11101101
                        +       1
                         11101110 = -18
```

Como se pode esperar, o resultado da dupla negação de um número é o próprio número:

```
                   -18 = 11101110 (complemento de dois)
complemento bit a bit  = 00010001
                        +       1
                         00010010 = +18
```

A validade da operação descrita acima pode ser demonstrada usando a definição da representação em complemento de dois dada pela Equação 8.2. Novamente, considere uma seqüência de n dígitos binários $a_{n-1}\, a_{n-2} \ldots a_1 a_0$ como um número inteiro A, representado em complemento de dois, de modo que seu valor seja:

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Obtenha agora o complemento bit a bit, $\overline{a_{n-1}}\ \overline{a_{n-2}} \ldots \overline{a_0}$, e, tratando o resultado como um número inteiro sem sinal, adicione 1. Finalmente, interprete a seqüência de n bits de dígitos binários resultante como um número inteiro B, representado em complemento de dois, de modo que seu valor seja:

$$B = -2^{n-1} \overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

É fácil mostrar que $A = -B$, ou seja, $A + B = 0$:

$$A + B = -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i(a_i + \overline{a_i})\right)$$
$$= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right)$$
$$= -2^{n-1} + 1 + (2^{n-1} - 1)$$
$$= -2^{n-1} + 2^{n-1} = 0$$

Na seqüência de equações acima, supomos, ao adicionar 1, que a seqüência de bits representa um número inteiro sem sinal (obtido fazendo o complemento bit a bit do número original) e tratamos o resultado, então, como um número inteiro em complemento de dois. Dois casos especiais devem ser considerados. O primeiro é se $A = 0$. Nesse caso, para uma representação de 8 bits, temos:

```
                         0 = 00000000 (complemento de dois)
  complemento bit a bit = 11111111
                         +        1
                        100000000 = 0
```

O bit 'vai-um' (*carry-in*) com valor 1 obtido na posição mais à esquerda (indicado por um dígito sombreado) é ignorado. Como resultado, temos que o complemento de dois de 0 é 0, como deveria ser.

O segundo caso especial é mais problemático. Se negarmos o padrão de bits constituído de um bit com valor 1 seguido de $n - 1$ bits de valor 0, obteremos esse mesmo número. Por exemplo, para uma palavra de 8 bits, temos:

```
                        -128 = 10000000 (complemento de dois)
 complemento de bit a bit = 01111111
                            +        1
                             10000000  =  -128
```

Essa anomalia não pode ser evitada. O número de padrões de bits distintos de uma palavra de n bits é 2^n, que é um número par. Desejamos representar números inteiros positivos, negativos e 0. Se a quantidade de números inteiros positivos e negativos que podem ser representados for a mesma (sinal-magnitude), então existirão duas representações para 0. Se existir apenas uma representação para 0 (complemento de dois), então as quantidades de números positivos e negativos que podem ser representados serão diferentes. No caso da representação em complemento de dois, usando uma palavra de n–bits, haverá uma representação para o valor -2^n, mas não para $+2^n$.

```
 1001                   1100
+0101                  +0100
 1110 = -2             10000 = 0

(a)  (-7) + (+5)       (b)  (-4) + (+4)
```

```
 0011                   1100
+0100                  +1111
 0111 = 7              11011 = -5

(c)  (+3) + (+4)       (d)  (-4) + (-1)
```

```
 0101                   1001
+0100                  +1010
 1001 = Overflow       10011 = Overflow

(e)  (+5) + (+4)       (f)  (-7) + (-6)
```

Figura 8.4 Adição de números na representação em complemento de dois.

Adição e subtração

A adição de números na representação em complemento de dois é apresentada na Figura 8.4. Os quatro primeiros exemplos mostram operações bem-sucedidas. Se o resultado da operação for positivo, será obtido um número positivo na notação binária. Se for negativo, será obtido um número negativo em complemento de dois. Note que, em alguns dos exemplos, ocorre um 'vai-um' para fora do bit mais significativo da palavra, que é ignorado.

O resultado de uma adição pode ter um número de bits maior do que o tamanho da palavra usada. Essa condição é denominada ***overflow***. Quando ocorre *overflow*, a ULA deve sinalizar esse fato, para que o resultado não seja utilizado. A detecção de *overflow* é feita de acordo com a seguinte regra: na adição de dois números, ambos positivos ou negativos, ocorrerá *overflow* somente se o resultado tiver sinal oposto. As Figuras 8.4e e 8.4f mostram exemplos de *overflow*. Note que pode ocorrer *overflow* mesmo não havendo 'vai-um' para fora do bit mais significativo.

A subtração também é implementada facilmente, usando a seguinte regra: para subtrair um número S (subtraendo) de um número M (minuendo), pegue o complemento de dois (negação) de S e acrescente esse valor a M. Dessa maneira, a subtração é implementada usando a adição, como indicado na Figura 8.5. Os dois últimos exemplos mostram que pode ocorrer *overflow*.

Voltando à Figura 8.2, note que, para números com n bits, podemos subtrair um valor k positivo (ou adicionar um valor k negativo) movendo $2^n - k$ posições no sentido horário. Note que $2^n - k$ é o complemento de dois de k. Isso demonstra graficamente que a subtração $M - S$ pode ser feita somando a M o complemento de dois de S.

```
    0010                 0101
   +1001                +1110
    1011 = -5           10011 = 3

(a) M = 2 = 0010     (b) M = 5 = 0101
    S = 7 = 0111         S = 2 = 0010
   -S =     1001        -S =     1110
```

```
    1011                 0101
   +1110                +0010
   11001 = -7            0111 = 7

(c) M = -5 = 1011    (d) M =  5 = 0101
    S =  2 = 0010        S = -2 = 1110
   -S =      1110       -S =      0010
```

```
    0111                 1010
   +0111                +1100
    1110 = Overflow     10110 = Overflow

(e) M =  7 = 0111    (f) M = -6 = 1010
    S = -7 = 1001        S =  4 = 0100
   -S =      0111       -S =      1100
```

Figura 8.5 Subtração de números na representação em complemento de dois ($M - S$).

A Figura 8.6 sugere os caminhos de dados e elementos de hardware necessários para efetuar a adição e a subtração. O elemento central é um somador binário (ou meio-somador), que recebe dois números e produz como resultado a soma desses números e uma indicação de *overflow*. O somador binário trata os dois operandos como números inteiros sem sinal. (Um circuito lógico que implementa o somador é mostrado no Apêndice A.) Esses dois operandos são apresentados ao somador em dois registradores, designados, nesse caso, como registradores *A* e *B*. O resultado pode ser armazenado em um desses registradores ou em um terceiro. A ocorrência de *overflow* é indicada por um bit de *overflow* (0 = não ocorreu *overflow*; 1 = ocorreu *overflow*). Na operação de subtração, o subtraendo (registrador *B*) é passado por um circuito que calcula seu complemento de dois, sendo esse valor, então, passado para o somador.

Multiplicação

Comparada às operações de adição e subtração, a multiplicação é uma operação complexa, seja implementada em hardware seja em software. Uma grande variedade de algoritmos de multiplicação tem sido usada em diversos computadores. O propósito desta subseção é dar ao leitor uma noção do tipo de abordagem tipicamente adotada. Começamos pelo problema mais simples de multiplicar dois números inteiros sem sinal (não negativos), para depois examinar uma das técnicas mais usadas para a multiplicação de números em complemento de dois.

Números inteiros sem sinal

A Figura 8.7 mostra a multiplicação de números inteiros binários sem sinal, tal como é feita com lápis e papel. Diversas observações importantes podem ser feitas:

1. A multiplicação envolve a geração de produtos parciais, um para cada dígito do multiplicador. Esses produtos parciais são somados para a obtenção do produto final.
2. Os produtos parciais são determinados facilmente. Quando o bit do multiplicador é 0, o produto parcial é 0. Quando é 1, o produto parcial é o próprio multiplicando.
3. O produto total é obtido somando-se os produtos parciais. Para isso, cada produto parcial sucessivo é deslocado um dígito para a esquerda, em relação ao produto parcial anterior.
4. A multiplicação de números inteiros binários de n bits resulta em um produto com até $2n$ bits de comprimento.

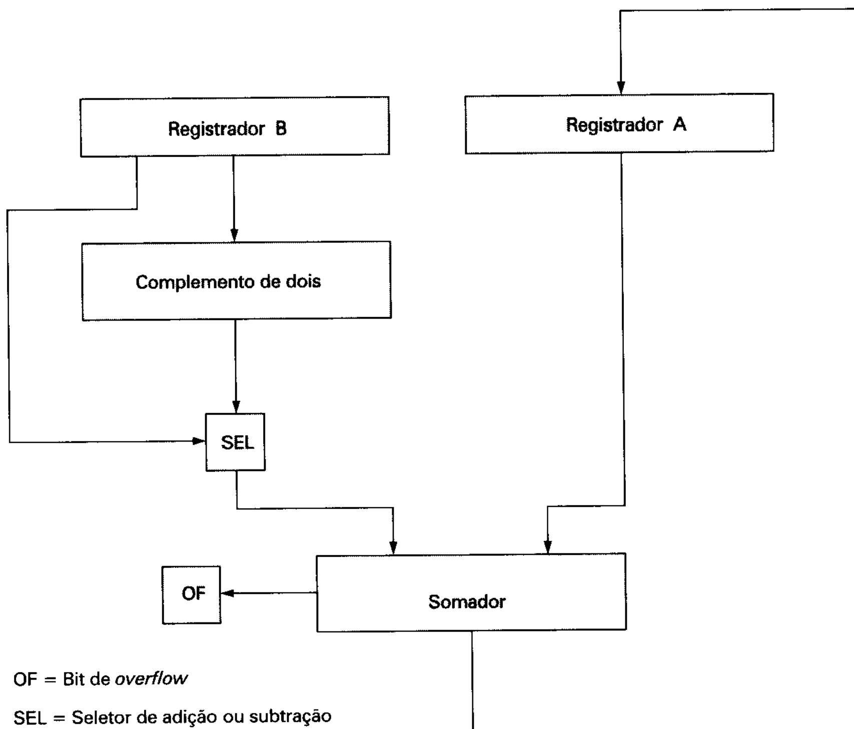


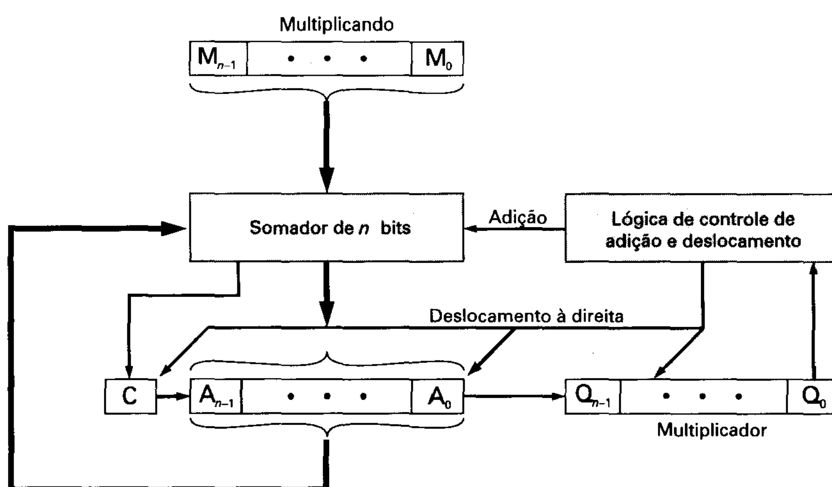
Figura 8.6 Diagrama de blocos do hardware de adição e subtração.

```
     1011      Multiplicando (11)
   × 1101      Multiplicador (13)
   ------
     1011  \
    0000    |  Produtos
   1011     |  parciais
  1011     /
 --------
 10001111        Produto (143)
```

Figura 8.7 Multiplicação de números inteiros binários sem sinal.

Multiplicações podem ser feitas de modo mais eficiente do que na forma usual, em que são feitas usando-se lápis e papel. Primeiro, podemos acumular imediatamente cada produto parcial obtido, em vez de esperar o cálculo de todos os produtos parciais. Isso elimina a necessidade de armazenar todos os produtos parciais; é preciso, dessa maneira, um número menor de registradores. Segundo, podemos poupar algum tempo na geração de produtos parciais. Para cada 1 no multiplicador, é necessário realizar uma operação de soma e um deslocamento; para cada 0, apenas um deslocamento é necessário.

A Figura 8.8a mostra uma implementação possível que emprega essas idéias. O multiplicador e o multiplicando são carregados em dois registradores (Q e M). Também é necessário um terceiro registrador, o registrador A, que é inicializado com valor 0. Existe ainda um registrador C, de 1 bit, inicializado com 0, que contém um potencial bit 'vai-um' resultante da adição.



(a) Diagrama de blocos

C	A	Q	M		
0	0000	1101	1011	Valores iniciais	
0	1011	1101	1011	Adição	Primeiro
0	0101	1110	1011	Deslocamento	Ciclo
0	0010	1111	1011	Deslocamento	Segundo Ciclo
0	1101	1111	1011	Adição	Terceiro
0	0110	1111	1011	Deslocamento	Ciclo
1	0001	1111	1011	Adição	Quarto
0	1000	1111	1011	Deslocamento	Ciclo

(b) Exemplo da Figura 8.7 (Produto em A, Q)

Figura 8.8 Implementação do hardware de multiplicação de números binários sem sinal.

A operação do multiplicador se dá como a seguir. A lógica de controle lê os bits do multiplicador, um de cada vez. Se Q_0 for 1, o multiplicador será adicionado ao registrador A e o resultado, armazenado nesse registrador, sendo o bit C usado para indicar a ocorrência de

overflow. Então, todos os bits dos registradores *C*, *A* e *Q* são deslocados um bit para a direita, de modo que o bit C vá para A_{n-1}, A_0 vá para Q_{n-1} e Q_0 seja perdido. Se Q_0 é 0, então nenhuma adição é efetuada, sendo feito apenas o deslocamento dos bits. Esse processo é repetido para cada bit do multiplicador original. O produto de $2n$ bits resultante estará contido nos registradores *A* e *Q*. Um fluxograma dessa operação é mostrado na Figura 8.9 e um exemplo é dado na Figura 8.8b. Note por meio desse exemplo, que, no segundo ciclo, quando o bit do multiplicador é 0, não é feita a adição.

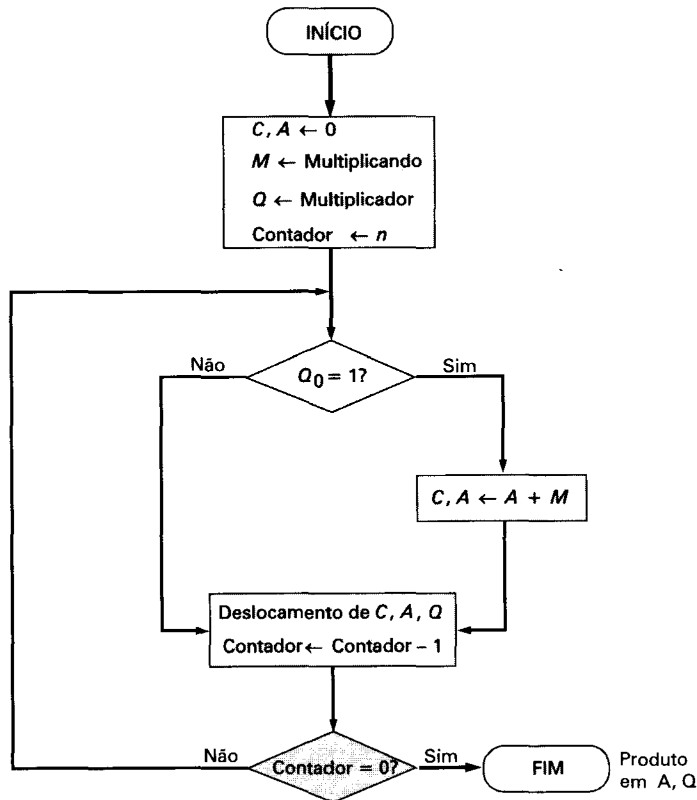


Figura 8.9 Fluxograma da multiplicação de números binários sem sinal.

Multiplicação de números em complemento de dois

Vimos que é possível fazer adição e subtração de números na notação em complemento de dois tratando-os como números inteiros sem sinal. Considere o seguinte:

```
 1001
+0011
 1100
```

Se esses números forem considerados números inteiros sem sinal, então estaremos fazendo a adição 9 (1001) mais 3 (0011) para obter 12 (1100). Como os números são representados em complementos de dois, estamos de fato fazendo a adição –7 (1001) mais 3 (0011) e obtendo –4 (1100).

Infelizmente, esse esquema simples não funciona para a multiplicação. Para mostrar isso, consideramos novamente a Figura 8.7. Multiplicando 11 (1011) por 13 (1101), obtemos 143 (10001111). Se interpretarmos esses números como números em complemento de dois, teremos –5 (1011) vezes –3 (1101), que é igual a –113 (10001111). Esse exemplo mostra que a multiplicação direta não funciona se o multiplicando e o multiplicador são negativos. De fato, ela não funciona nos casos em que o multiplicando ou o multiplicador é negativo. Para ver isso, voltamos à Figura 8.7 para explicar o que está sendo feito em termos de operações com potências de 2. Lembre-se de que qualquer número binário sem sinal pode ser expresso como uma soma de potências de 2. Portanto:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 2^3 + 2^2 + 2^0$$

```
        1011
      × 1101
    --------
    00001011   1011 × 1 × 2^0
    00000000   1011 × 0 × 2^1
    00101100   1011 × 1 × 2^2
    01011000   1011 × 1 × 2^3
    --------
    10001111
```

Figura 8.10 Multiplicação de dois números inteiros de 4 bits sem sinal produzindo um resultado de 8 bits.

A multiplicação de um número binário por 2^n é feita deslocando esse número n bits para a esquerda. Com isso em mente, a Figura 8.10 reproduz o exemplo da Figura 8.7, mostrando a geração dos produtos parciais por multiplicação explícita. A única diferença na Figura 8.10 é reconhecer que os produtos parciais devem ser vistos como números de $2n$ bits, gerados a partir de um multiplicando de n bits.

Dessa maneira, o multiplicando 1011 de 4 bits é armazenado como um número inteiro sem sinal em uma palavra de 8 bits, ou seja, como 00001011. Cada produto parcial (exceto o correspondente a 2^0) é constituído desse número deslocado para a esquerda, com as posições não ocupadas à direita preenchidas com zeros (por exemplo, um deslocamento para a esquerda de duas posições produz 00101100).

Podemos agora mostrar que a multiplicação direta não funciona no caso em que o multiplicando é negativo. O problema é que cada contribuição do multiplicando negativo como produto parcial deve ser um número negativo de $2n$ bits; os bits de sinal dos produtos parciais devem ser alinhados. Isso pode ser visto na Figura 8.11, que mostra a multiplicação de 1001 por 0011. Se esses números são tratados como números inteiros sem sinal, a multiplicação $9 \times 3 = 27$ prossegue da forma usual. No entanto, se 1001 for interpretado como o número em complemento de dois que representa o valor –7, então cada produto parcial deverá ser um número negativo, em complemento de dois, de $2n$ (8) bits, como mostrado na Figura 8.11b. Note que isso pode ser feito estendendo cada produto parcial para a esquerda, com bits de valor 1.

É fácil perceber que a multiplicação direta também não funciona se o multiplicador é negativo. A razão é que, nesse caso, os bits do multiplicador não correspondem aos deslocamentos e às multiplicações que devem ocorrer. Por exemplo, considere o número decimal –3, escrito em

complemento de dois com 4 bits como 1101. Se calcularmos os produtos parciais simplesmente com base no valor de cada posição de bit, obteremos a seguinte correspondência:

$$1101 \leftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

1001	(9)
× 0011	(3)
00001001	1001×2^0
00010010	1001×2^1
00011011	(27)

(a) Números inteiros sem sinal

1001	(−7)
× 0011	(3)
11111001	$(-7) \times 2^0 = (-7)$
11110010	$(-7) \times 2^1 = (-14)$
11101011	(−21)

(b) Números inteiros em complemento de dois

Figura 8.11 Comparação entre multiplicação de números inteiros sem sinal e em complemento de dois.

De fato, o que se deseja obter é $-(2^1 + 2^0)$. Portanto, o multiplicador não pode ser usado diretamente da maneira como descrevemos.

Existem diversas soluções possíveis para esse dilema. Uma delas seria converter o multiplicador e o multiplicando para números positivos, efetuar a multiplicação e, então, caso o sinal dos dois números originais seja diferente, tomar o complemento de dois do resultado. Os projetistas têm preferido usar técnicas que não requeiram esse passo final de transformação. Um dos algoritmos mais usados é o algoritmo de Booth. Esse algoritmo tem também a vantagem de efetuar a multiplicação de maneira mais rápida do que em uma abordagem mais direta.

O algoritmo de Booth é representado na Figura 8.12 e pode ser descrito como a seguir. Como antes, multiplicador e multiplicando são armazenados nos registradores Q e M, respectivamente. Existe também um registrador de 1 bit, posicionado logicamente à direita do bit menos significativo (Q_0) do registrador Q e designado como Q_{-1}, cujo uso é explicado a seguir. O resultado da multiplicação é dado nos registradores A e Q. A e Q_{-1} são inicializados com valor 0. Como antes, a lógica de controle examina os bits do multiplicador, um de cada vez. Quando cada bit é examinado, também é examinado o bit à sua direita. Se esses dois bits forem iguais (1–1 ou 0–0), então todos os bits dos registradores A, Q e Q_{-1} serão deslocados 1 bit para a direita. Se eles forem diferentes, o multiplicando será somado ou subtraído do registrador A, dependendo se os dois bits são 0–1 ou 1–0, respectivamente. Após essa operação de adição ou subtração, ocorre o deslocamento de um bit para a direita, que é feito de tal maneira que o bit mais à esquerda de A, denominado A_{n-1}, é deslocado para A_{n-2}, mas também permanece em A_{n-1}. Isso é necessário para preservar o sinal do número armazenado em A e Q. Esse deslocamento é conhecido como deslocamento aritmético, porque preserva o bit de sinal.

A Figura 8.13 mostra a seqüência de eventos do algoritmo de Booth para o caso da multiplicação de 7 por 3. A mesma operação é representada de maneira mais compacta na Figura 8.14a. O restante da Figura 8.14 apresenta outros exemplos de uso do algoritmo. Como se pode ver, o algoritmo funciona com qualquer combinação de números positivos e negativos. Note também a eficiência do algoritmo. Blocos de 1s ou de 0s são ignorados, sendo feita, em média, apenas uma adição ou subtração por bloco.

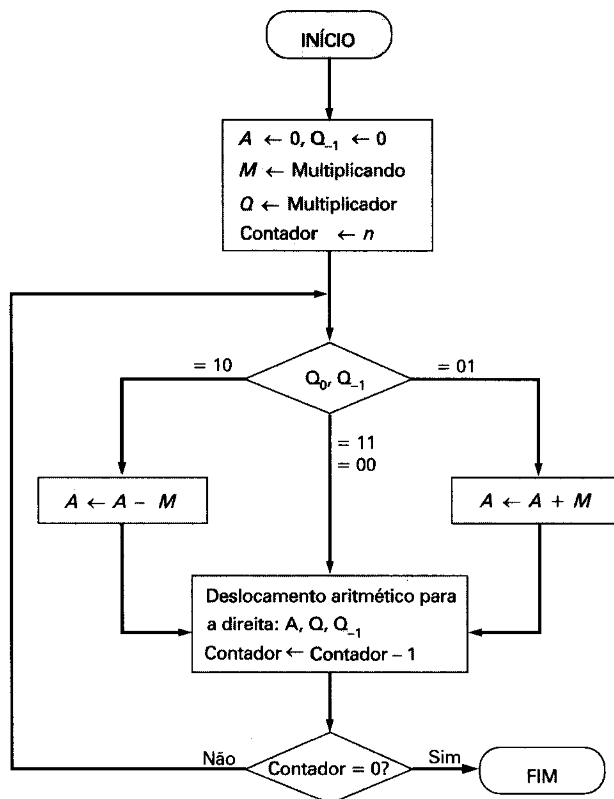


Figura 8.12 Algoritmo de Booth para a multiplicação em complementos de dois.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Valores Iniciais	
1001	0011	0	0111	$A \leftarrow A - M$	Primeiro Ciclo
1100	1001	1	0111	Deslocamento	
1110	0100	1	0111	Deslocamento	Segundo Ciclo
0101	0100	1	0111	$A \leftarrow A + M$	Terceiro Ciclo
0010	1010	0	0111	Deslocamento	
0001	0101	0	0111	Deslocamento	Quarto Ciclo

Figura 8.13 Exemplo do algoritmo de Booth.

```
     0111                 0111
  × 0011    (0)        × 1101    (0)
11111001    1-0        11111001  1-0
0000000     1-1        0000111   0-1
000111      0-1        111001    1-0
00010101    (21)       11101011  (-21)
 (a) (7) × (3) = (21) (b) (7) × (-3) = (-21)

     1001                 1001
  × 0011    (0)        × 1101    (0)
00000111    1-0        00000111  1-0
0000000     1-1        1111001   0-1
111001      0-1        000111    1-0
11101011    (-21)      00010101  (21)
 (c) (-7) × (3) = (-21)(d) (-7) × (-3) = (21)
```

Figura 8.14 Exemplos de uso do algoritmo de Booth.

Por que o algoritmo de Booth funciona? Considere primeiramente o caso em que o multiplicador é positivo. Em particular, suponha que o multiplicador seja constituído de um bloco de 1s cercado por 0s (por exemplo, 00011110). Como sabemos, a multiplicação pode ser feita somando-se cópias apropriadamente deslocadas do multiplicando:

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

O número de operações requeridas pode ser reduzido para dois se observarmos que:

$$2^n + 2^{n-1} + \ldots + 2^{n-K} = 2^{n+1} - 2^{n-K} \quad (8.3)$$

Portanto,

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

Assim, o produto pode ser gerado efetuando-se apenas uma adição e uma subtração do multiplicando. Esse esquema pode ser estendido para qualquer número de blocos de 1s do multiplicador, incluindo o caso em que o bloco tem um único 1. Portanto,

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

O algoritmo de Booth opera de acordo com esse esquema, efetuando uma subtração quando é encontrado o primeiro 1 de um bloco (1 – 0) e uma adição quando é encontrado o fim do bloco (0 – 1).

Para mostrar que esse esquema também funciona se o multiplicador for negativo, observe o seguinte. Seja X um número negativo na notação em complemento de dois:

$$\text{Representação de } X = \{1x_{n-2}x_{n-3} \ldots x_1x_0\}$$

Então, o valor de X pode ser expresso como:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0) \quad (8.4)$$

Isso pode ser verificado aplicando o algoritmo aos números da Tabela 8.2.

O bit mais à esquerda de X é 1, uma vez que X é negativo. Suponha que o bit de valor 0, mais à esquerda, esteja na posição k. Então, X é da forma:

$$\text{Representação de } X = \{111 \ldots 10x_{k-1}\, x_{k-2} \ldots x_1x_0\} \quad (8.5)$$

E o valor de X é:

$$X = -2^{n-1} + 2^{n-2} + \ldots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \ldots + (x_0 \times 2^0) \quad (8.6)$$

Da Equação 8.3 obtemos que:

$$2^{n-2} + 2^{n-3} + \ldots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Reordenando os termos, temos:

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^{k+1} = -2^{k+1} \quad (8.7)$$

Substituindo a Equação 8.7 na Equação 8.6, obtemos:

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \ldots + (x_0 \times 2^0) \quad (8.8)$$

Por fim, retornamos ao algoritmo de Booth. Tendo em vista a representação de X (Equação 8.5), é claro que todos os bits desde x_0 até o bit de valor 0 mais à esquerda são manipulados de maneira adequada, pois produzem todos os termos da Equação 8.8, exceto (-2^{k+1}), e assim estão na forma correta. À medida que o algoritmo examina o bit 0 mais à esquerda e encontra o próximo 1 (2^{k+1}), ocorre uma transição 1 – 0 e é realizada uma subtração (-2^{k+1}). Esse é o termo restante da Equação 8.8.

Como exemplo, considere a multiplicação de um número M por (–6). Na representação em complemento de dois, com palavras de 8 bits, (–6) é representado como 11111010. Da Equação 8.4, sabemos que:

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

o que você pode verificar facilmente. Portanto,

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Usando a Equação 8.7, obtemos:

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

que você pode verificar que também é igual a $M \times (-6)$. Finalmente, seguindo nossa linha de raciocínio anterior, temos:

$$M \times (11111010) = M \times (-2^3 + 2^2 - 2^1)$$

Podemos perceber que o algoritmo de Booth opera de acordo com o esquema anterior. Ele efetua uma subtração quando o primeiro 1 é encontrado (1 – 0), uma adição quando é encontrada uma transição (0 – 1) e, finalmente, outra subtração quando o primeiro 1 do próximo bloco de 1s é encontrado. Portanto, o algoritmo de Booth efetua menos adições e subtrações do que um algoritmo mais direto.

Divisão

A divisão é, de certa maneira, mais complexa que a multiplicação, embora seja baseada nos mesmos princípios gerais. Como antes, a base para o algoritmo é a abordagem usada ao efetuar a operação com lápis e papel, e a operação envolve repetidas execuções de adição, subtração e deslocamento.

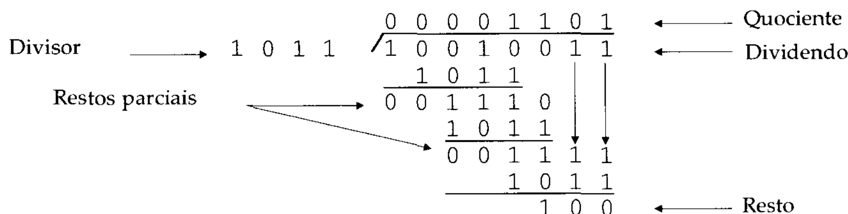


Figura 8.15 Divisão de números inteiros binários sem sinal.

A Figura 8.15 mostra um exemplo de divisão de números inteiros binários sem sinal. É instrutivo descrever o processo em detalhes. Primeiramente, os bits do dividendo são examinados, da esquerda para a direita, até que se obtenha um conjunto de bits que represente um número maior ou igual ao divisor. Quando esse número é encontrado, diz-se que o divisor *divide* o número. Enquanto isso não ocorre, são introduzidos 0s no quociente, da esquerda para a direita. Quando esse evento ocorre, é colocado um 1 no quociente e o divisor é subtraído do dividendo parcial. O resultado é conhecido como *resto parcial*. Desse ponto em diante, a divisão segue um padrão cíclico. A cada ciclo, bits adicionais do dividendo são anexados ao resto parcial, até que o resultado seja maior ou igual ao divisor. Como antes, o divisor é subtraído desse número, para produzir um novo resto parcial. O processo continua até que todos os bits do dividendo tenham sido examinados.

A Figura 8.16 mostra um algoritmo de máquina para o longo processo de divisão. O divisor é colocado no registrador M e o dividendo, no registrador Q. A cada passo, os registradores A e Q, juntos, são deslocados um bit para a esquerda. M é subtraído de A, para determinar se A divide o resto parcial[1]. Se dividir, então o valor do bit Q_0 será 1. Caso contrário, o valor de Q_0 será 0 e o de M será somado a A, para restaurar seu valor anterior. O contador então é decrementado e o processo é repetido por n passos. Ao final, o quociente estará no registrador Q e o resto, no registrador A.

Esse processo pode, com alguma dificuldade, ser estendido para números negativos. Uma possível abordagem para números em complemento de dois é apresentada a seguir. Diversos exemplos dessa abordagem são mostrados na Figura 8.17. O algoritmo pode ser descrito como a seguir:

1. Carregar o divisor no registrador M e o dividendo nos registradores A e Q. O dividendo deve ser expresso como um número em complemento de dois com $2n$ bits. Por exemplo, o número 0111 de 4 bits seria representado como 00000111 e o número 1001, como 11111001.
2. Deslocar o conteúdo dos registradores A e Q, juntos, um bit para a esquerda.

1. Essa é uma subtração de números inteiros sem sinal. Se for requerido empréstimo de uma unidade para o bit mais significativo (bit 'empresta-um'), o resultado será negativo.

3. Se M e A têm o mesmo sinal, fazer $A \leftarrow A - M$; caso contrário, $A \leftarrow A + M$.
4. A operação anterior será bem-sucedida se o sinal de A for o mesmo, antes e depois da operação.
   a. Se a operação for bem-sucedida ou se ($A = 0$ e $Q = 0$), então faça $Q_0 \leftarrow 1$.
   b. Se a operação não for bem-sucedida e se ($A \neq 0$ ou $Q \neq 0$), então faça $Q_0 \leftarrow 0$ e restaure o antigo valor de A (somando M a A).
5. Repita os passos 2 a 4 enquanto houver bits a examinar em Q.
6. Ao final, o resto estará em A. Se o divisor e o dividendo tiverem o mesmo sinal, o quociente estará em Q; caso contrário, o quociente correto é o complemento de dois do número armazenado em Q.

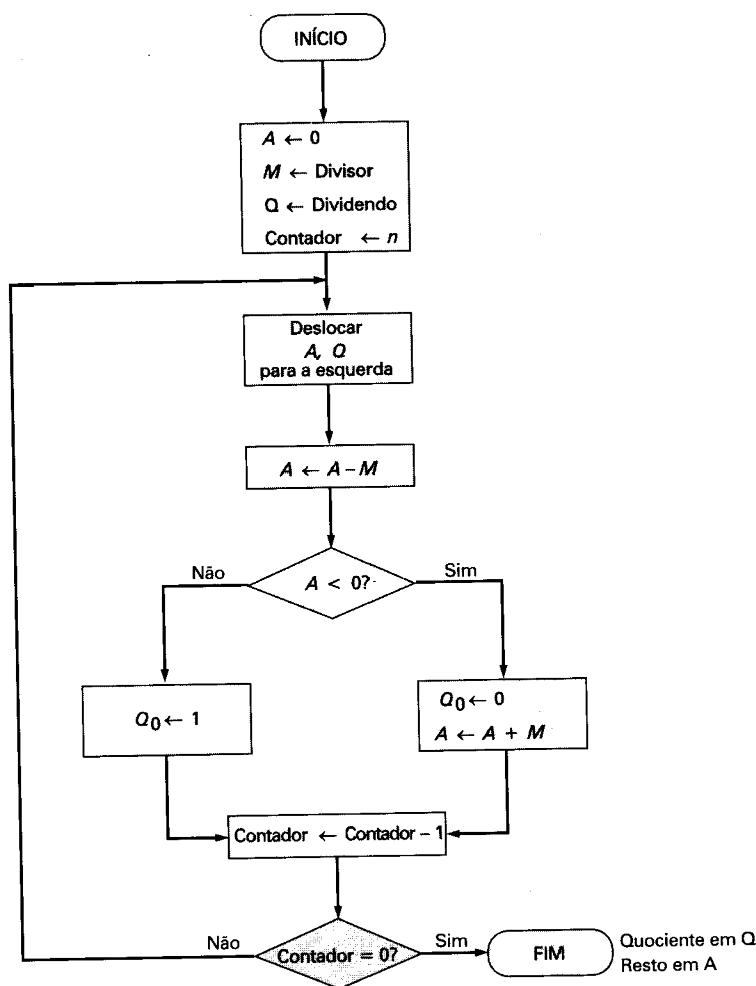


Figura 8.16 Fluxograma da divisão de números binários sem sinal.

Você deve notar, a partir da Figura 8.17, que (–7) ÷ (3) e (7) ÷ (–3) produzem restos diferentes. Isso porque o resto é definido por:

$$D = Q \times V + R$$

onde:

D = dividendo
Q = quociente
V = divisor
R = resto

Os resultados mostrados na Figura 8.17 são coerentes com essa fórmula.

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Valor inicial	0000	0111	Valor inicial
0000	1110	Deslocar	0000	1110	Deslocar
1101		Subtrair	1101		Adicionar
0000	1110	Restaurar	0000	1110	Restaurar
0001	1100	Deslocar	0001	1100	Deslocar
1110		Subtrair	1110		Adicionar
0001	1100	Restaurar	0001	1100	Restaurar
0011	1000	Deslocar	0011	1000	Deslocar
0000		Subtrair	0000		Adicionar
0000	1001	Fazer $Q_0 = 1$	0000	1001	Fazer $Q_0 = 1$
0001	0010	Deslocar	0001	0010	Deslocar
1110		Subtrair	1110		Adicionar
0001	0010	Restaurar	0001	0010	Restaurar

(a) (7) ÷ (3) (b) (7) ÷ (–3)

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Valor inicial	1111	1001	Valor inicial
1111	0010	Deslocar	1111	0010	Deslocar
0010		Adicionar	0010		Subtrair
1111	0010	Restaurar	1111	0010	Restaurar
1110	0100	Deslocar	1110	0100	Deslocar
0001		Adicionar	0001		Subtrair
1110	0100	Restaurar	1110	0100	Restaurar
1100	1000	Deslocar	1100	1000	Deslocar
1111		Adicionar	1111		Subtrair
1111	1001	Fazer $Q_0 = 1$	1111	1001	Fazer $Q_0 = 1$
1111	0010	Deslocar	1111	0010	Deslocar
0010		Adicionar	0010		Subtrair
1111	0010	Restaurar	1111	0010	Restaurar

(c) (–7) ÷ (3) (d) (–7) ÷ (–3)

Figura 8.17 Exemplos de divisão em complemento de dois.

8.4 REPRESENTAÇÃO DE NÚMEROS DE PONTO FLUTUANTE

Princípios

Usando uma notação de ponto fixo (por exemplo, complemento de dois), é possível representar certa faixa de números inteiros positivos e negativos, centrada em 0. Esse formato permite também a representação de números com parte fracionária, bastando fixar uma posição adequada para a vírgula que separa a parte inteira e a parte fracionária.

Essa abordagem tem, entretanto, algumas limitações. Ela não possibilita representar números muito grandes nem frações muito pequenas. Além disso, em uma divisão de dois números muito grandes, a parte fracionária do quociente pode ser perdida.

Para números decimais, essa limitação é superada com o uso da notação científica. Por exemplo, 976.000.000.000.000 pode ser representado como $9{,}76 \times 10^{14}$ e 0,0000000000000976, como $9{,}76 \times 10^{-14}$. A vírgula é deslizada dinamicamente para uma posição conveniente e é usado um expoente de 10 adequado, para representar o mesmo valor original. Isso possibilita expressar números muito grandes e muito pequenos com poucos dígitos.

A mesma abordagem pode ser usada para números binários. Um número pode ser representado na forma:

$$\pm M \times B^{\pm E}$$

Esse número pode ser armazenado em uma palavra binária, com três campos:

- Sinal: mais ou menos
- Mantissa M
- Expoente E

A base *B* é implícita e não precisa ser armazenada porque é a mesma para todos os números.

Os princípios usados na representação de números binários de ponto flutuante podem ser bem explicados por meio de um exemplo. A Figura 8.18a mostra um formato típico de números de ponto flutuante de 32 bits. O bit mais à esquerda armazena o sinal do número (0 = positivo; 1 = negativo). O valor do expoente é armazenado nos 8 bits seguintes. A representação usada é conhecida como representação polarizada. Um valor fixo, chamado de polarização, é subtraído ao valor desse campo para se obter o verdadeiro valor do expoente. Tipicamente, a polarização é igual a $(2^{k-1} - 1)$, onde k é o número de bits do expoente binário. Nesse caso, o campo de 8 bits pode conter números de 0 a 255. Com uma polarização igual a 127, os verdadeiros valores dos expoentes estão compreendidos na faixa de −127 a +128. Nesse exemplo, considera-se que a base é 2.

A Tabela 8.2 mostra a representação polarizada para números inteiros de 4 bits. Note que, se os bits de uma representação polarizada forem tratados como números inteiros sem sinal, a magnitude relativa dos números não mudará. Por exemplo, nas duas representações, tanto polarizada quanto sem sinal, o maior número é 1111 e o menor número, 0000. Isso não é verdadeiro para as representações sinal-magnitude, em complemento de dois e em complemento de um. Uma vantagem da representação polarizada é que, para fins de comparação, números de ponto flutuante não negativos podem ser tratados como números inteiros.

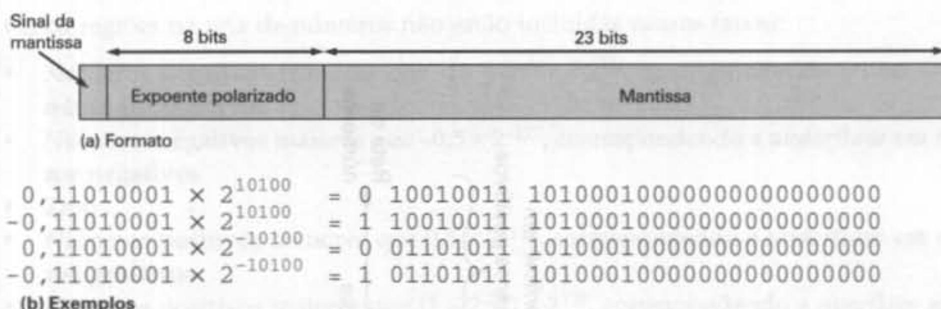


Figura 8.18 Formato típico da representação de números de ponto flutuante de 32 bits.

A parte final da palavra (23 bits nesse caso) é a mantissa. Note que um mesmo número de ponto flutuante pode ser expresso de muitas maneiras. Por exemplo, as seguintes representações são equivalentes, em que a mantissa é expressa na forma binária:

$$0{,}110 \times 2^5$$
$$110 \times 2^2$$
$$0{,}0110 \times 2^6$$

Para simplificar as operações sobre números de ponto flutuante, tipicamente é requerido que esses números estejam normalizados. No nosso exemplo, um número normalizado diferente de zero tem a forma:

$$\pm 0{,}1\,bbb\ldots b \times 2^{\pm E}$$

onde *b* é um dígito binário (0 ou 1). Isso implica que o bit mais à esquerda da mantissa é sempre 1. Por isso, não é necessário armazenar esse bit, pois ele é implícito. Portanto, o campo de 23 bits é usado para armazenar uma mantissa de 24 bits com valor entre 0,5 e 1,0.

A Figura 8.18b mostra alguns exemplos de números armazenados nesse formato. Note as seguintes características:

- O sinal é armazenado no primeiro bit da palavra.
- O primeiro bit da mantissa verdadeira é sempre 1 e não precisa ser armazenado no campo de mantissa.
- O valor 127 é adicionado ao expoente verdadeiro para ser armazenado no campo de expoente.
- A base é 2.

A Figura 8.19 indica a faixa de números que podem ser representados em uma palavra de 32 bits, usando essa representação. Na notação em complemento de dois, podem ser representados todos os números inteiros de -2^{31} a $2^{31}-1$, ou seja, um total de 2^{32} números distintos. No formato de ponto flutuante da Figura 8.18, números nas seguintes faixas podem ser representados:

- Números negativos entre $-(1 - 2^{-24}) \times 2^{128}$ e $-0{,}5 \times 2^{-127}$
- Números positivos entre $0{,}5 \times 2^{-127}$ e $(1 - 2^{-24}) \times 2^{128}$

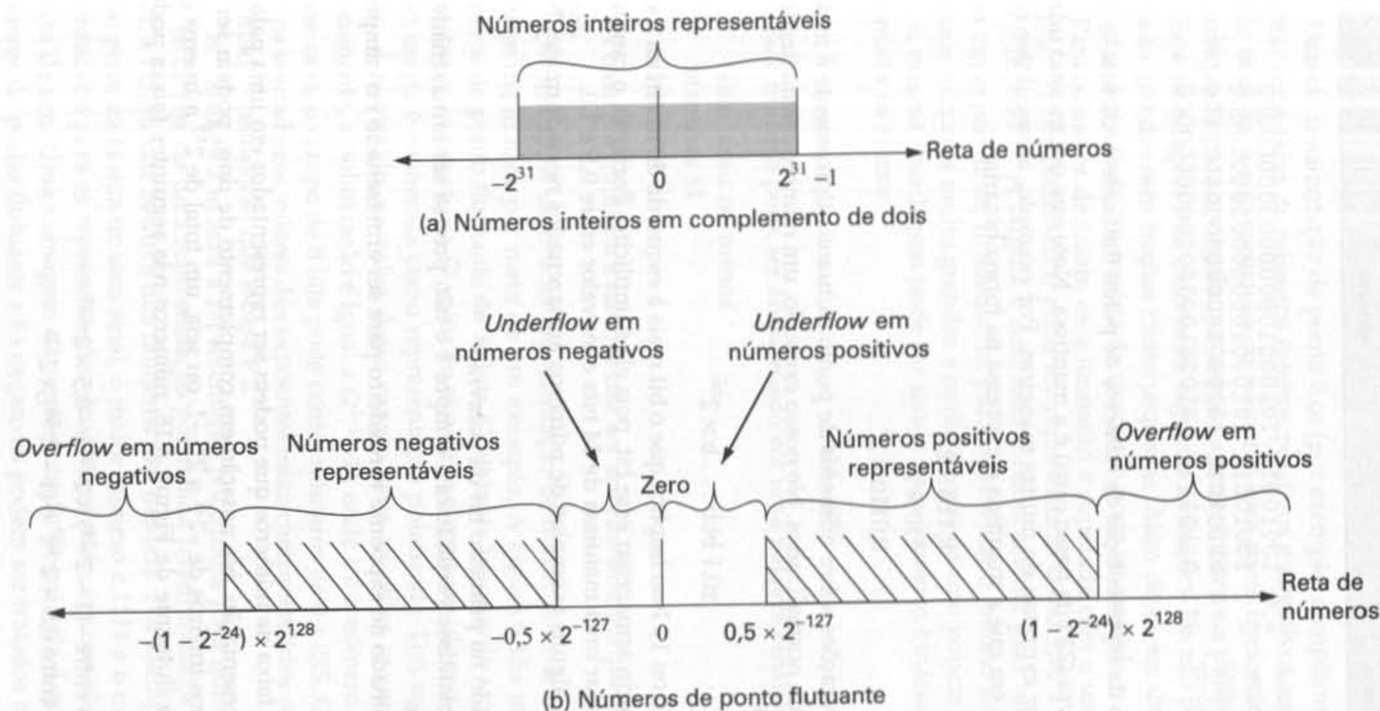


Figura 8.19 Números representáveis em formato típico de 32 bits.

Cinco regiões na reta de números não estão incluídas nessas faixas:

- Números negativos menores que $-(1 - 2^{-24}) \times 2^{128}$, correspondendo a ***overflow* em números negativos**
- Números negativos maiores que $-0,5 \times 2^{-127}$, correspondendo a ***underflow* em números negativos**
- Zero
- Números positivos menores que $0,5 \times 2^{-127}$, correspondendo a ***underflow* em números positivos**
- Números positivos maiores que $(1 - 2^{-24}) \times 2^{128}$, correspondendo a ***overflow* em números positivos**

Essa representação, tal como apresentada, não acomoda o valor 0. Entretanto, como veremos, representações de ponto flutuante na verdade incluem um padrão de bits especial para designar zero. Em uma operação aritmética, ocorre *overflow* quando a magnitude do resultado é maior do que o maior valor que pode ser expresso com expoente igual a 128 (por exemplo, $2^{120} \times 2^{100} = 2^{220}$). Ocorre *underflow* quando a magnitude é muito pequena (por exemplo, $2^{-120} \times 2^{-100} = 2^{-220}$). A ocorrência de *underflow* é um problema menos sério, pois o resultado geralmente pode ser aproximado satisfatoriamente por 0.

É importante observar que a notação de ponto flutuante não possibilita representar um número maior de valores distintos. O número máximo de valores distintos que podem ser representados com 32 bits continua sendo 2^{32}. Entretanto, esses números são divididos em duas faixas, uma positiva e uma negativa.

Veja também que os números representados na notação de ponto flutuante não estão igualmente distribuídos ao longo da reta de números, como é o caso de números de ponto fixo. Existe uma quantidade maior de valores representáveis próximo à origem, e essa quantidade diminui com a distância à origem, como é mostrado na Figura 8.20. Esse é um dos problemas da aritmética de ponto flutuante: muitos cálculos produzem resultados não exatos, que têm de ser arredondados para o valor mais próximo que a notação possibilita representar.

No tipo de formato mostrado na Figura 8.18, existe um conflito entre a faixa de valores e a precisão dos números representáveis. O exemplo mostra que são reservados 8 bits para o expoente e 23 bits para a mantissa. Se aumentarmos o número de bits do expoente, expandimos a faixa de valores representáveis. Entretanto, como apenas um número fixo de valores distintos pode ser expresso, reduzimos a densidade desses números, o que diminui a precisão. A única maneira de aumentar tanto o alcance quanto a precisão é usar um maior número de bits. Por isso, a maioria dos computadores oferece, pelo menos, números de precisão simples e números de precisão dupla. Por exemplo, um formato de precisão simples pode ter 32 bits e um formato de precisão dupla, 64 bits.

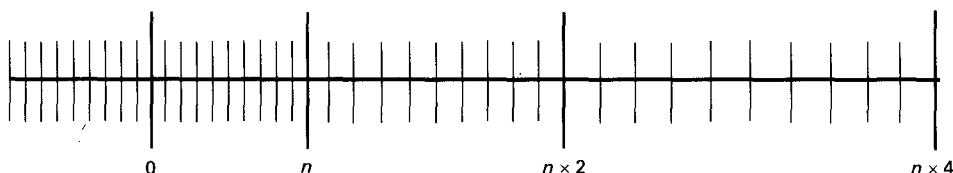


Figura 8.20 Densidade de números de ponto flutuante.

Portanto, existe um conflito entre o número de bits reservados para o expoente e para a mantissa. Essa questão é, porém, ainda mais complicada. A base subentendida para o expoente não precisa, necessariamente, ser 2. Na arquitetura IBM S/390, por exemplo, é usada a base 16. O formato é composto de um expoente de 7 bits e uma mantissa de 24 bits. Assim, por exemplo, temos:

$$0{,}11010001 \times 2^{10100} = 0{,}11010001 \times 16^{101}$$

onde o expoente armazenado representa 5, em vez de 20.

A vantagem de usar um expoente maior é que se pode representar uma faixa maior de números, utilizando o mesmo número de bits para o expoente. Lembre-se, entretanto, de que o número de valores distintos que podem ser representados permanece o mesmo. Portanto, para um formato de tamanho fixo, uma base de expoente maior permite um maior alcance (faixa de valores) ao custo de uma menor precisão.

Padrão IEEE para representação de números binários de ponto flutuante

A mais importante representação de ponto flutuante é definida no padrão IEEE 754 (IEEE, 1985). Esse padrão foi desenvolvido para facilitar a portabilidade de programas entre processadores, além de encorajar o desenvolvimento de programas de processamento numérico sofisticados. Ele tem sido largamente adotado, sendo usado em quase todos os processadores e co-processadores aritméticos modernos.

O padrão IEEE define um formato simples de 32 bits e um formato duplo de 64 bits (Figura 8.21), com expoentes de 8 e 11 bits, respectivamente. A base implícita é 2. Na adição, o padrão define dois formatos estendidos, simples e duplo, cujo formato exato é dependente de implementação. Os formatos estendidos incluem bits adicionais no expoente (alcance estendido) e na mantissa (precisão estendida). Eles são usados para cálculos intermediários. O uso desses formatos diminui a chance de o resultado final ser contaminado por excessivo erro de arredondamento, uma vez que fornecem maior precisão. Além disso, como permitem maior alcance, também diminui a chance de *overflow* em operações intermediárias, o que poderia abortar uma computação cujo resultado final seria representável no formato básico. Uma motivação adicional para o formato estendido simples é que ele possui alguns dos benefícios do formato duplo, sem incorrer na penalidade de tempo normalmente associada a uma precisão mais alta. A Tabela 8.3 resume as características dos quatro formatos.

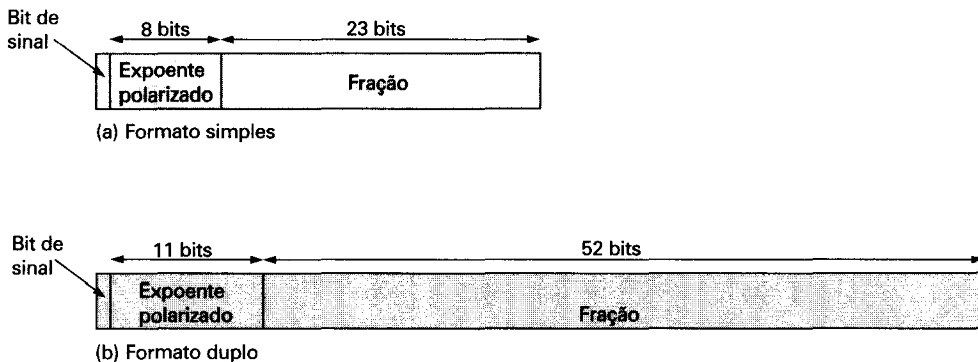


Figura 8.21 Formatos do padrão IEEE 754.

Nem todos os padrões de bits do formato IEEE são interpretados da maneira usual; alguns são usados para representar valores especiais. A Tabela 8.4 indica os valores atribuídos a vários padrões de bits. Os casos em que o expoente é totalmente preenchido com zeros (0) ou com uns (255 no formato simples e 2047 no formato duplo) definem valores especiais. As seguintes classes de números são representadas:

- Se o intervalo de valores do expoente é de 1 a 254 no formato simples ou de 1 a 2.046 no formato duplo, são representados números de ponto flutuante normalizados, diferentes de zero. O expoente é polarizado, de modo que a faixa de valores de expoente seja –126 a +127 no formato simples e –1022 a +1023 no formato duplo. Um número normalizado tem o bit 1 à esquerda da vírgula fracionária igual a 1; esse bit é implícito, o que dá uma mantissa (chamada fração na descrição do padrão IEEE) efetiva de 24 bits no formato simples ou de 53 bits no formato duplo.
- Um expoente igual a zero junto com uma fração igual a zero representa zero, positivo ou negativo, dependendo do bit de sinal. Como mencionamos anteriormente, é útil ter uma representação para o valor exato de 0.
- Um expoente totalmente preenchido com uns junto com uma fração igual a zero representa infinito, positivo ou negativo, dependendo do bit de sinal. É também muito útil ter uma representação para infinito. Isso possibilita ao usuário decidir entre tratar *overflow* como uma condição de erro ou propagar esse valor (dando prosseguimento à execução do programa).
- Um expoente de valor zero junto com uma fração diferente de zero representa um número não-normalizado. Nesse caso, o bit à esquerda da vírgula fracionária é zero e o expoente verdadeiro é –126 no formato simples ou –1022 no formato duplo. O número é positivo ou negativo conforme o bit de sinal.
- Um expoente totalmente preenchido com uns junto com uma fração diferente de zero representa o valor NaN (*Not a Number*), que significa *Não é um número*, e é usado para sinalizar diversas condições de exceção.

Tabela 8.3 Parâmetros do formato IEEE 754

	Formato			
Parâmetro	**Simples**	**Simples estendido**	**Duplo**	**Duplo estendido**
Tamanho da palavra (bits)	32	≥43	64	≥79
Tamanho do expoente (bits)	8	≥11	11	≥15
Polarização do expoente	127	Não especificado	1023	Não especificado
Expoente máximo	127	≥1023	1023	≥16383
Expoente mínimo	–126	≤–1022	–1022	≤–16382
Faixa de números (base 10)	10^{-38}, 10^{+38}	Não especificado	10^{-308}, 10^{+308}	Não especificado
Tamanho da mantissa (bits)*	23	≥31	52	≥63
Número de expoentes	254	Não especificado	2046	Não especificado
Número de frações	2^{23}	Não especificado	2^{52}	Não especificado
Número de valores	$1{,}98 \times 2^{31}$	Não especificado	$1{,}99 \times 2^{63}$	Não especificado

* Não inclui o bit implícito

O significado de números não-normalizados e de NaNs é discutido na Seção 8.5.

Tabela 8.4 Interpretação de números de ponto flutuante no padrão IEEE 754

	Precisão simples (32 bits)				**Precisão dupla (64 bits)**			
	Sinal	**Expoente polarizado**	**Fração**	**Valor**	**Sinal**	**Expoente polarizado**	**Fração**	**Valor**
Zero positivo	0	0	0	0	0	0	0	0
Zero negativo	1	0	0	–0	1	0	0	–0
Infinito positivo	0	255 (todos 1s)	0	∞	0	2047 (todos 1s)	0	∞
Infinito negativo	1	255 (todos 1s)	0	$-\infty$	1	2047 (todos 1s)	0	$-\infty$
NaN silencioso	0 ou 1	255 (todos 1s)	$\neq 0$	NaN	0 ou 1	2047 (todos 1s)	$\neq 0$	NaN
NaN sinalizador	0 ou 1	255 (todos 1s)	$\neq 0$	NaN	0 ou 1	2047 (todos 1s)	$\neq 0$	NaN
Diferente de zero, normalizado positivo	0	$0 < e < 255$	f	$2^{e-127}(1,f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1,f)$
Diferente de zero, normalizado negativo	1	$0 < e < 255$	f	$-2^{e-127}(1,f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1,f)$
Não-normalizado positivo	0	0	$f \neq 0$	$2^{e-126}(0,f)$	0	0	$f \neq 0$	$2^{e-1022}(0,f)$
Não-normalizado negativo	1	0	$f \neq 0$	$-2^{e-126}(0,f)$	1	0	$f \neq 0$	$-2^{e-1022}(0,f)$

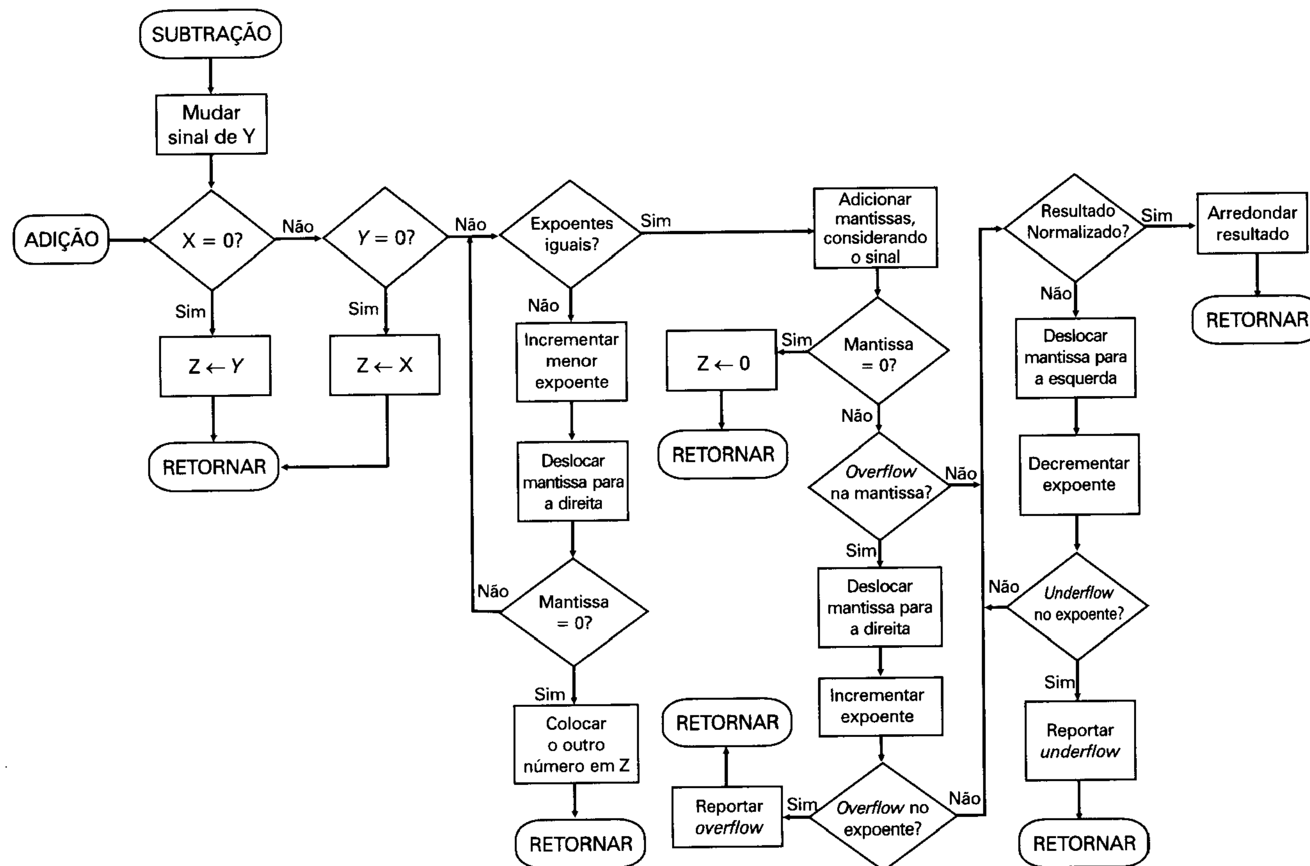


Figura 8.22 Adição e subtração de ponto flutuante ($Z \leftarrow X \pm Y$).

8.5 ARITMÉTICA DE NÚMEROS DE PONTO FLUTUANTE

A Tabela 8.5 relaciona as operações básicas da aritmética de ponto flutuante. Nas operações de adição e subtração, é necessário assegurar que os dois operandos tenham o mesmo valor de expoente, o que pode requerer o deslocamento da vírgula fracionária em um dos operandos para se obter o alinhamento apropriado. A multiplicação e a divisão são operações mais diretas.

Alguns problemas podem ocorrer como resultado dessas operações:

- ***Overflow* no expoente:** um expoente positivo excede o maior valor possível para um expoente. Em alguns sistemas, isso pode ser designado como $+\infty$ ou $-\infty$.
- ***Underflow* no expoente:** um expoente negativo é menor que o menor valor possível para um expoente. Isso significa que o número é muito pequeno para ser representado e pode ser tratado como sendo 0.
- ***Underflow* na mantissa:** no processo de alinhamento das mantissas, pode ocorrer o transbordo de dígitos para fora da extremidade da direita da mantissa. Como discutiremos a seguir, isso requer alguma forma de arredondamento.
- ***Overflow* na mantissa:** a adição de duas mantissas de mesmo sinal pode resultar em um 'vai-um' do bit mais significativo. Isso pode ser corrigido com um novo alinhamento do número, como é explicado a seguir.

Tabela 8.5 Números de ponto flutuante e operações aritméticas

Números de ponto flutuante	Operações aritméticas
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Exemplos:
$X = 0,3 \times 10^2 = 30$
$Y = 0,2 \times 10^3 = 200$
$X + Y = (0,3 \times 10^{2-3} + 0,2) \times 10^3 = 0,23 \times 10^3 = 230$
$X - Y = (0,3 \times 10^{2-3} - 0,2) \times 10^3 = (-0,17) \times 10^3 = -170$
$X \times Y = (0,3 \times 0,2) \times 10^{2+3} = 0,06 \times 10^5 = 6000$
$X \div Y = (0,3 \div 0,2) \times 10^{2-3} = 1,5 \times 10^{-1} = 0,15$

Adição e subtração

Na aritmética de ponto flutuante, a adição e a subtração são operações mais complexas que a multiplicação e a divisão. Isso se deve à necessidade de alinhar os operandos, de modo que torne seus expoentes iguais. O algoritmo de adição e de subtração tem quatro fases básicas:

1. Verificar se algum operando é zero.
2. Alinhar as mantissas.
3. Adicionar ou subtrair as mantissas.
4. Normalizar o resultado.

Um fluxograma típico dessas operações é mostrado na Figura 8.22. Uma descrição passo a passo realça as principais funções requeridas na adição e na subtração de números de ponto flutuante. Considere um formato semelhante ao da Figura 8.21. Em uma operação de adição ou subtração, os dois operandos devem ser armazenados em registradores usados pela ULA. Se o formato de ponto flutuante inclui um bit de mantissa implícito, esse bit deve ser tornado explícito no momento de efetuar a operação.

Como a adição e a subtração são idênticas, exceto por uma mudança de sinal, o processo começa pela mudança de sinal do subtraendo, caso a operação seja uma subtração. A seguir, se o operando for 0, o resultado será o valor do outro operando.

A próxima fase é manipular os números, de modo que seus expoentes se tornem iguais. Para ver por que isso é necessário, considere a seguinte adição de números decimais:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Claramente, não podemos apenas adicionar as mantissas. Antes disso, os dígitos devem ser colocados em posições equivalentes, ou seja, o 4 do segundo número deve ser alinhado com o 3 do primeiro número. Dessa maneira, os dois expoentes serão iguais, que é a condição matemática para que dois números possam ser somados. Portanto,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4{,}56 \times 10^0) = 127{,}56 \times 10^0$$

O alinhamento pode ser obtido tanto deslocando o número menor para a direita (aumentando seu expoente) quanto deslocando o número maior para a esquerda. Como ambas as operações podem resultar em perda de dígitos, o deslocamento é efetuado no número de menor expoente; qualquer dígito perdido tem, portanto, um valor relativamente menor. O alinhamento é feito deslocando os dígitos da mantissa para a direita e incrementando o expoente, repetidamente, até que os dois expoentes sejam iguais. (Note que, se a base implícita for 16, o deslocamento de um dígito para a direita corresponderá a um deslocamento para a direita de 4 bits.) Caso esse processo resulte em valor 0 na mantissa, o resultado da operação é o outro número. Portanto, se os dois números tiverem expoentes que diferem significativamente, o menor número será perdido.

Em seguida, as duas mantissas são somadas, levando em conta seus sinais. Como os sinais podem ser diferentes, o resultado pode ser 0. Também existe a possibilidade de ocorrer *overflow* na mantissa por um dígito. Nesse caso, a mantissa do resultado é deslocada para a direita e o expoente é incrementado. Essa operação pode resultar em *overflow* no expoente, o que termina a operação, sendo reportado um erro.

A fase seguinte normaliza o resultado. A normalização consiste em deslocar os dígitos da mantissa para a esquerda, até que o dígito mais significativo (1 bit ou 4 bits, se a base for 16) seja diferente de zero. A cada deslocamento, o expoente é decrementado, podendo, portanto, ocorrer *underflow* no expoente. Finalmente, o resultado deve ser arredondado e, então, é reportado. O arredondamento será discutido após examinarmos as operações de multiplicação e divisão.

Multiplicação e divisão

A multiplicação e a divisão de ponto flutuante são operações muito mais simples do que a adição e a subtração, como mostramos a seguir.

Em princípio, consideramos a multiplicação, mostrada na Figura 8.23. Primeiramente, se qualquer dos operandos for 0, o resultado será 0. O próximo passo é somar os expoentes. Se os expoentes forem armazenados na forma polarizada, a soma dos expoentes dobrará a polarização. Portanto, o valor da polarização deve ser subtraído da soma dos expoentes. Poderá então ocorrer tanto *overflow* quanto *underflow* no expoente, o que termina o algoritmo, sendo reportado um erro.

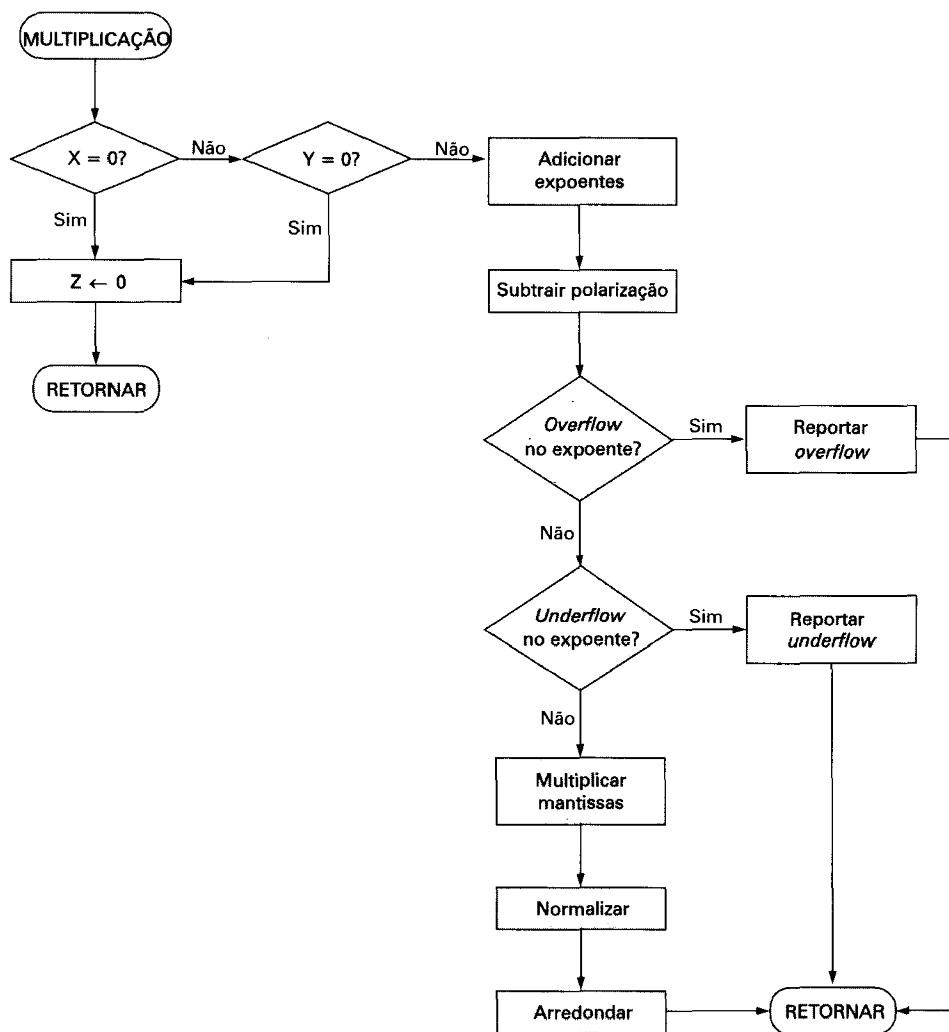


Figura 8.23 Multiplicação de números de ponto flutuante (Z ← X × Y).

Se o expoente do produto estiver dentro da faixa de números representáveis, o passo seguinte será multiplicar as mantissas, levando em conta seus sinais. A multiplicação é feita

do mesmo modo que para números inteiros. Nesse caso, estamos tratando de uma representação sinal-magnitude, mas os detalhes são semelhantes aos da operação sobre números em complemento de dois. O produto terá o dobro do comprimento do multiplicando e do multiplicador. Bits extras são perdidos com o arredondamento.

Depois de calculado o produto, o resultado é então normalizado e arredondado, tal como é feito na adição e na subtração. A normalização pode resultar em um *underflow* no expoente.

Finalmente, considere o fluxograma para a divisão, representado na Figura 8.24. Também aqui, o primeiro passo é testar se algum dos operandos é 0. Caso o divisor seja 0, é reportado um erro ou o resultado é a representação de infinito, dependendo da implementação. Caso o dividendo seja 0, o resultado será 0. Se não ocorrer nenhum desses casos, no passo seguinte, o expoente do divisor será subtraído do expoente do dividendo. Isso remove a polarização, que deve, portanto, ser adicionada ao resultado. Em seguida, são efetuados testes de *overflow* e *underflow* no expoente.

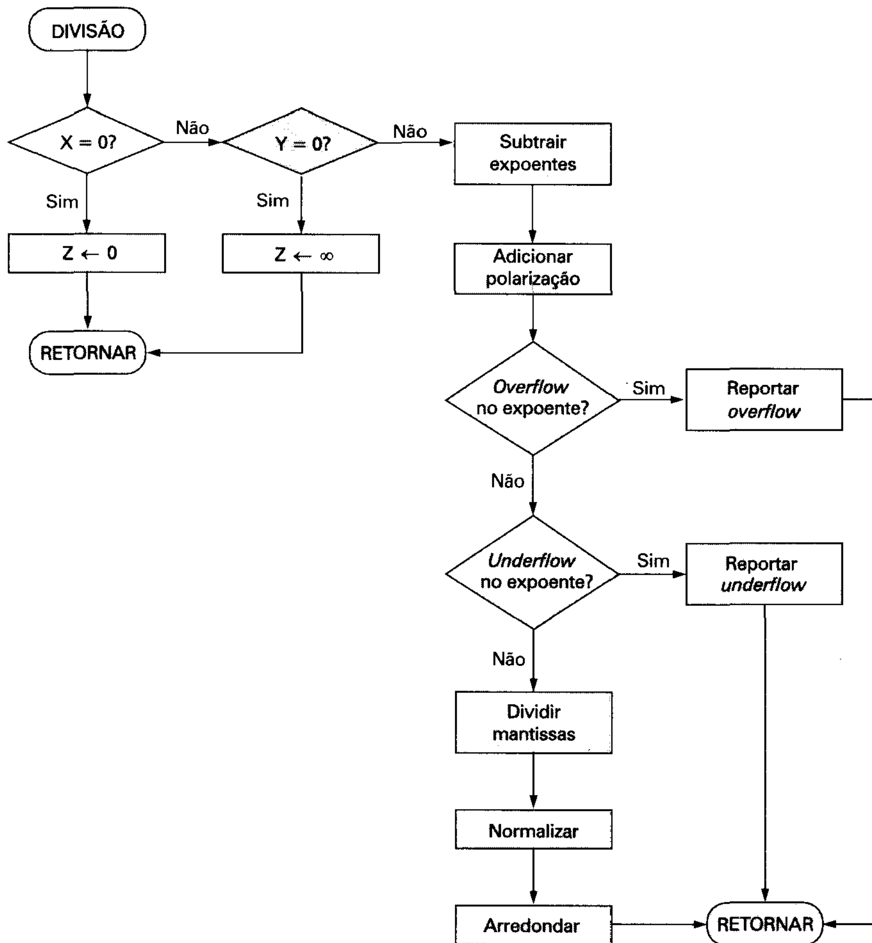


Figura 8.24 Divisão de números de ponto flutuante (Z ← X/Y).

O passo seguinte é dividir as mantissas. Esse passo é seguido pela normalização e pelo arredondamento do resultado.

Considerações de precisão

Bits de guarda

Mencionamos anteriormente que antes de uma operação de ponto flutuante o expoente e a mantissa são carregados em registradores da ULA. No caso da mantissa, o tamanho do registrador é quase sempre maior que o número de bits da mantissa mais o bit implícito. O registrador contém bits adicionais, chamados bits de guarda, que são usados para preencher os bits extras até a extremidade direita da mantissa com zeros.

A razão do uso desses bits é mostrada na Figura 8.25. Considere os números no formato IEEE, cuja mantissa tem 24 bits, incluindo um bit 1 implícito, à esquerda da vírgula fracionária. Dois números de valor próximo são, por exemplo, $X = 1{,}00 \ldots 00 \times 2^1$ e $Y = 1{,}11 \ldots 11 \times 2^0$. Se o número menor for subtraído do maior, ele deverá ser deslocado um bit para a direita, para alinhar os expoentes. Isso é mostrado na Figura 8.25a. Nesse processo, Y perde um bit; o resultado é 2^{-22}. A mesma operação é repetida, na parte (b), adicionando agora bits de guarda. Nesse caso, o bit menos significativo não é perdido no alinhamento e o resultado é 2^{-23}, ou seja, uma diferença de um fator 2 com relação à resposta anterior. Se a base for 16, a perda de precisão poderá ser ainda maior. Como mostram as Figuras 8.25c e 8.25d, a diferença pode ser um fator igual a 16.

$$\begin{array}{rcl} x & = & 1{,}000 \;\ldots\ldots\; 00 \times 2^{1} \\ -y & = & \underline{0{,}111 \;\ldots\ldots\; 11} \times 2^{1} \\ z & = & 0{,}000 \;\ldots\ldots\; 01 \times 2^{1} \\ & = & 1{,}000 \;\ldots\ldots\; 00 \times 2^{-22} \end{array}$$

(a) Exemplo binário, sem bits de guarda

$$\begin{array}{rcl} x & = & 1{,}000 \;\ldots\ldots\; 00\;0000 \times 2^{1} \\ -Y & = & \underline{0{,}111 \;\ldots\ldots\; 11\;1000} \times 2^{1} \\ z & = & 0{,}000 \;\ldots\ldots\; 00\;1000 \times 2^{1} \\ & = & 1{,}000 \;\ldots\ldots\; 00\;0000 \times 2^{-23} \end{array}$$

(b) Exemplo binário, com bits de guarda

$$\begin{array}{rcl} x & = & {,}100000 \times 16^{1} \\ -y & = & \underline{{,}0FFFFF} \times 16^{1} \\ z & = & {,}000001 \times 16^{1} \\ & = & {,}100000 \times 16^{-4} \end{array}$$

(c) Exemplo hexadecimal, sem bits de guarda

$$\begin{array}{rcl} x & = & {,}100000\;00 \times 16^{1} \\ -y & = & \underline{{,}0FFFFF\;F0} \times 16^{1} \\ z & = & {,}000000\;10 \times 16^{1} \\ & = & {,}100000\;00 \times 16^{-5} \end{array}$$

(d) Exemplo hexadecimal, com bits de guarda

Figura 8.25 Uso dos bits de guarda.

Arredondamento

Outro detalhe que afeta a precisão do resultado é a política de arredondamento. O resultado de qualquer operação sobre as mantissas geralmente é armazenado em um registrador de tamanho maior. Quando o resultado é colocado novamente no formato de ponto flutuante, os bits extras devem ser descartados.

Diversas técnicas para arredondamento foram exploradas. De fato, o padrão IEEE relaciona quatro abordagens alternativas:

- **Arredondar para o mais próximo:** o resultado é arredondado para o número representável mais próximo.
- **Arredondar para cima** (+ ∞)**:** o resultado é arredondado para cima, na direção de infinito positivo.
- **Arredondar para baixo** (– ∞)**:** o resultado é arredondado para baixo, na direção de infinito negativo.
- **Arredondar para 0:** o resultado é arredondado na direção de zero.

Cada uma dessas políticas é considerada separadamente a seguir. O modo padrão de arredondamento é **arredondar para o mais próximo**, que é definido como a seguir: o resultado é o valor representável mais próximo do resultado com precisão infinita. Se existirem dois valores representáveis igualmente próximos, o resultado será aquele cujo bit menos significativo for igual a 0.

Por exemplo, se os bits extras, além dos 23 que podem ser armazenados, forem 10010, então esses bits extras totalizarão mais que a metade da última posição de bit representável. Nesse caso, a resposta correta é obtida adicionando 1 ao último bit representável, isto é, arredondando para cima, para o número representável imediatamente maior. Suponha agora que os bits extras sejam 01111. Nesse caso, os bits extras totalizam menos que a metade da última posição de bit representável. A resposta correta é obtida simplesmente ignorando os bits extras (truncando), ou seja, arredondando para baixo, para o número representável imediatamente inferior.

O padrão também considera o caso especial em que os bits extras são da forma 10000... Aqui, o resultado está exatamente no meio de dois valores representáveis. Uma técnica possível nesse caso seria truncar sempre, uma vez que essa operação é a mais simples. No entanto, o problema dessa abordagem é que ela introduz uma polarização pequena, porém cumulativa, ao longo de uma seqüência de computações. O que é necessário é um método de arredondamento não-polarizado. Uma abordagem possível seria arredondar para cima ou para baixo, conforme um número aleatório, de modo que, em média, o resultado seria não-polarizado. O argumento contra essa abordagem é que ela não apresenta resultados previsíveis, determinísticos. A abordagem adotada pelo padrão IEEE é forçar que o resultado seja par: se o resultado de uma computação está exatamente no meio de dois números representáveis, o valor é arredondado para cima, caso o último bit representável seja 1, e será truncado, se o último bit for 0.

As duas opções seguintes, **arredondar para infinito positivo ou negativo**, são úteis na implementação de uma técnica conhecida como aritmética intervalar. A aritmética intervalar fornece um método para monitorar e controlar erros em computações de ponto flutuante, produzindo dois valores para cada resultado. Os dois valores correspondem aos limites inferior e superior de um intervalo que contém o resultado verdadeiro. A largura do intervalo, isto é,

a diferença entre o limite inferior e o limite superior, indica a precisão do resultado. Se os limites do intervalo não forem representáveis, então, os limites inferior e superior serão arredondados para baixo e para cima, respectivamente. Embora a largura do intervalo possa variar de acordo com a implementação, muitos algoritmos são projetados para produzir intervalos pequenos. Se a faixa de valores entre os limites do intervalo for suficientemente pequena, então o resultado obtido será suficientemente preciso. Caso contrário, esse fato ao menos é conhecido e análises adicionais podem ser efetuadas.

A última técnica especificada no padrão é **arredondar para zero**. De fato, isso consiste simplesmente em truncar o número: os bits extras são ignorados. Essa é, certamente, a técnica mais simples. No entanto, o resultado é que a magnitude dos valores truncados é sempre menor ou igual ao valor original mais preciso, o que introduz uma constante polarização em direção a zero na operação. Essa polarização é mais séria do que a que foi discutida anteriormente, pois afeta todas as operações em que existam bits extras diferentes de zero.

Padrão IEEE para aritmética de números binários de ponto flutuante

O padrão IEEE 754 vai além da simples definição de um formato, delineando também práticas e procedimentos específicos para que a aritmética de ponto flutuante produza resultados uniformes, previsíveis e independentes da plataforma de hardware. Um desses aspectos, o arredondamento, já foi discutido. Nesta subseção, três outros tópicos são discutidos: infinito, NaNs e números não-normalizados.

Infinito

A aritmética de infinito é tratada como caso-limite da aritmética real, com valores infinitos com a seguinte interpretação:

$$-\infty < (\text{cada número finito}) < +\infty$$

Com exceção dos casos especiais discutidos a seguir, qualquer operação aritmética envolvendo infinito produz o resultado óbvio. Por exemplo,

$$
\begin{array}{lll}
5 + (+\infty) = +\infty & \quad & 5 \div (+\infty) = +0 \\
5 - (+\infty) = -\infty & & (+\infty) + (+\infty) = +\infty \\
5 + (-\infty) = -\infty & & (-\infty) + (-\infty) = -\infty \\
5 - (-\infty) = +\infty & & (-\infty) - (+\infty) = -\infty \\
5 \times (+\infty) = +\infty & & (+\infty) - (-\infty) = +\infty
\end{array}
$$

NaN silencioso e NaN sinalizador

Um NaN é uma entidade simbólica, codificada no formato de números de ponto flutuante, que pode ser de dois tipos: sinalizador ou silencioso. Um NaN sinalizador (*signaling NaN*) gera uma exceção de operação inválida sempre que é usado como operando. Esses NaNs fornecem valores para variáveis não inicializadas e para extensões aritméticas não submetidas ao padrão. Um NaN silencioso (*quiet NaN*) se propaga por meio de quase toda operação aritmética sem gerar uma exceção. A Tabela 8.6 indica as operações que dão como resultado um NaN silencioso.

Note que ambos os tipos de NaN têm o mesmo formato geral: o campo de expoente preenchido totalmente com uns e uma fração diferente de zero. O padrão de bits real da fração

(diferente de zero) é dependente de implementação. Os valores armazenados na fração podem ser usados para distinguir NaNs silenciosos e NaNs sinalizadores, além de especificar condições de exceção particulares.

Números não-normalizados

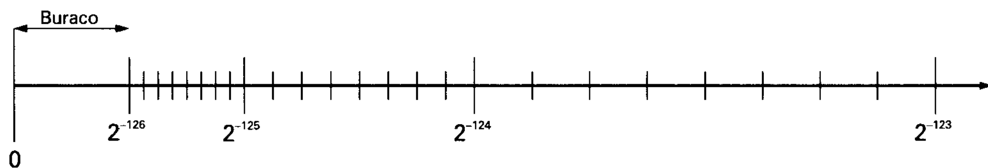
Números não-normalizados são incluídos no padrão IEEE 754 para manipular casos de *underflow* de expoente. Quando o expoente do resultado é muito pequeno (um expoente negativo com uma magnitude muito grande), o resultado torna-se não-normalizado, deslocando-se os dígitos da fração para a direita e incrementando o expoente a cada deslocamento, até que o expoente esteja dentro da faixa de valores representáveis.

A Figura 8.26 mostra o efeito da adição de números não-normalizados. Os números representáveis podem ser agrupados em intervalos da forma $[2^n, 2^{n+1}]$. Em cada um desses intervalos, o expoente do número permanece constante, enquanto a fração varia, produzindo uma distribuição uniforme de números representáveis dentro do intervalo. À medida que se aproxima de zero, cada intervalo sucessivo tem a metade da largura do intervalo anterior, mas contém a mesma quantidade de números representáveis. Por isso, a densidade de números representáveis aumenta conforme se aproxima de zero. Entretanto, se forem usados apenas os números normalizados, existirá um buraco entre o menor número normalizado e o zero. No caso do formato IEEE 754 de 32 bits, existem 2^{23} números representáveis em cada intervalo e o menor número positivo representável é 2^{-126}. Com a adição de números não-normalizados, 2^{23} números adicionais são uniformemente adicionados entre 0 e 2^{-126}.

O uso de números não-normalizados é conhecido como *underflow gradual* (Coonen, 1981). Sem o uso de números não-normalizados, o buraco entre o menor número não nulo representável e o zero é muito mais largo que o buraco entre o menor número não nulo representável e o número imediatamente maior. O *underflow* gradual preenche esse buraco, reduzindo o impacto do *underflow* de expoente a um nível comparável ao do arredondamento de números normalizados.

Tabela 8.6 Operações que produzem NaN silencioso

Operação	**NaN silencioso produzido por**
Qualquer	Qualquer operação sobre um NaN sinalizador
Adição ou subtração	Subtração de números de magnitude infinita $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiplicação	$0 \times \infty$
Divisão	$\frac{0}{0}$ ou $\frac{\infty}{\infty}$
Resto	x REM 0 ou ∞ REM y
Raiz quadrada	$\sqrt{x}$ onde $x < 0$



(a) Formato de 32 bits sem números não-normalizados



(b) Formato de 32 bits com números não-normalizados

Figura 8.26 Efeito de números não-normalizados no padrão IEEE 754.

8.6 LEITURA E SITE WEB RECOMENDADOS

Para o estudante interessado em aritmética computacional, uma referência indispensável diz respeito aos dois volumes de Swartzlander (1990). O primeiro volume foi originalmente publicado em 1980 e contém artigos fundamentais (alguns muito difíceis de serem obtidos de outras fontes) sobre os fundamentos da aritmética computacional. O segundo contém artigos mais recentes, cobrindo aspectos teóricos, de projeto e de implementação de aritmética computacional. Outros bons livros sobre aritmética computacional podem ser citados: Omondi (1994), Koren (1993) e Morgan (1992).

Para aritmética de números de ponto flutuante, o título de Goldberg (1991) é bastante adequado: "What every computer scientist should know about floating-point arithmetic" ["O que todo cientista de computação deveria saber sobre aritmética de ponto flutuante"]. Uma abordagem mais avançada é encontrada em Wallis (1990). Outra excelente abordagem sobre o tópico é encontrada em Knuth (1998), que cobre também a aritmética computacional de números inteiros. As seguintes abordagens, mais aprofundadas, são também importantes: Oberman e Flynn (1997a), Oberman e Flynn (1997b) e Soderquist e Leeser (1996).



Site Web recomendado:

- **IEEE 754:** contém os documentos do padrão IEEE 754, artigos e publicações relacionados e um conjunto de endereços úteis relativos a aritmética computacional.

8.7 EXERCÍCIOS

8.1 Uma outra representação de números inteiros binários que é algumas vezes utilizada é a representação em complemento de um. Números inteiros positivos são representados

da mesma maneira que na representação sinal-magnitude. Um número inteiro negativo é representado tomando-se o complemento booleano de cada bit do número positivo correspondente.

a. Forneça uma definição para números em complemento de um, usando uma soma ponderada de bits, semelhante às Equações 8.1 e 8.2.

b. Qual é a faixa de números que podem ser representados em complemento de um?

c. Defina um algoritmo para efetuar a adição em aritmética de complemento de um.

8.2 Adicione à Tabela 8.1 colunas correspondentes às representações sinal-magnitude e em complemento de um.

8.3 Considere a seguinte operação sobre uma palavra binária. Comece pelo bit menos significativo. Copie todos os bits que são iguais a 0, até que seja encontrado o primeiro bit diferente de zero, e copie também esse bit. Então, tome o complemento de cada bit daí por diante. Qual é o resultado?

8.4 Na Seção 8.3, a operação de complemento de dois é definida como a seguir. Para encontrar o complemento de dois de X, tome o complemento booleano de cada bit de X e então adicione 1.

a. Mostre que a seguinte definição é equivalente. Para um número inteiro *X* de *n* bits, o complemento de dois de *X* é formado tratando X como um número inteiro sem sinal e calculando $(2^n - X)$.

b. Mostre que a Figura 8.2 pode ser usada para verificar graficamente a equivalência mostrada em (a), indicando como um movimento no sentido horário pode ser usado para obter o resultado de uma subtração.

8.5 Calcule as seguintes subtrações usando aritmética de complemento de dois:

a. `111000` `-110011`	**b.** `11001100` `- 101110`	**c.** `111100001111` `-110011110011`	**d.** `11000011` `-11101000`

8.6 Verifique se é válida a seguinte definição alternativa para *overflow* em uma operação aritmética na representação em complemento de dois.
Se o resultado da operação ou-exclusivo dos bits 'vai-um' (*carry-in*) e 'vem-um' (*carry-out*) da coluna mais à esquerda for 1, então ocorreu *overflow*. Caso contrário, não ocorreu.

8.7 Compare as Figuras 8.9 e 8.12. Por que o bit C não é usado na Figura 8.12?

8.8 Dados $x = 0101$ e $y = 1010$, na notação em complemento de dois (isto é, $x = 5$ e $y = -6$), calcule o resultado de $p = x \times y$ usando o algoritmo de Booth.

8.9 Prove que uma multiplicação de dois números de *n* dígitos na base *B* fornece um produto de no máximo 2*n* dígitos.

8.10 Verifique a validade do algoritmo de divisão de números binários sem sinal apresentado na Figura 8.16, mostrando os passos envolvidos no cálculo da divisão feita na Figura 8.15. Use uma apresentação semelhante à da Figura 8.17.

8.11 O algoritmo de divisão de números inteiros em complemento de dois, descrito na Seção 8.3, é conhecido como um método de restauração, porque o valor no registrador *A* deve ser restaurado após uma subtração malsucedida. Uma abordagem ligeiramente mais complexa, conhecida como não-restauração, evita adições e subtrações desnecessárias. Proponha um algoritmo para essa última abordagem.

8.12 Na aritmética computacional de números inteiros, o quociente *J*/*K* da divisão de um número inteiro *J* por um número inteiro *K* é menor ou igual ao quociente usual. Verdadeiro ou falso?

8.13 Divida –145 por 13, na notação binária em complemento de dois, usando palavras de 12 bits. Use o algoritmo descrito na Seção 8.3.

8.14 Suponha que o expoente e deva ter valor no intervalo $0 \leq e \leq X$, com uma polarização q, e que a base seja b e a mantissa tenha p dígitos.

a. Quais são o maior e o menor valor positivos que podem ser representados?

b. Quais são o maior e o menor valor positivos que podem ser representados usando números de ponto flutuante normalizados?

8.15 Expresse os seguintes números em formato de ponto flutuante IEEE de 32 bits:

a. –5 **c.** –1,5 **e.** 1/16
b. –6 **d.** 384 **f.** –1/32

8.16 Expresse os seguintes números no formato de ponto flutuante de 32 bits da IBM, que usa um expoente de 7 bits, com uma base implícita igual a 16:

a. 1,0 **c.** 1/64 **e.** –15,0 **g.** $7{,}2 \times 10^{75}$
b. 0,5 **d.** 0,0 **f.** $5{,}4 \times 10^{-79}$

8.17 Qual seria o valor da polarização para um:

a. Expoente de base 2 (B = 2) em um campo de 6 bits?

b. Expoente de base 8 (B = 8) em um campo de 7 bits?

8.18 Desenhe uma reta de números semelhante à da Figura 8.19b para o formato de números de ponto flutuante da Figura 8.21b.

8.19 Considere um formato de ponto flutuante em que o expoente polarizado tem 8 bits e a mantissa, 23 bits. Qual é o padrão de bits dos seguintes números nesse formato:

a. –720

b. 0,645

8.20 Quando as pessoas falam sobre imprecisão em operações aritméticas sobre números de ponto flutuante, sempre atribuem erros ao cancelamento de bits que ocorre na subtração de valores muito próximos. Mas, quando X e Y são aproximadamente iguais, a diferença $X - Y$ é obtida precisamente, sem erro. O que essas pessoas realmente querem dizer?

8.21 Qualquer representação de números de ponto flutuante usada em um computador pode representar exatamente apenas certo conjunto de números reais; todos os demais valores devem ser aproximados. Se A' é o valor armazenado que se aproxima do valor real A, então o erro relativo, r, é expresso como:

$$r = \frac{A - A'}{A}$$

Represente o valor decimal +0,4 no seguinte formato de ponto flutuante: base = 2, expoente polarizado de 4 bits e mantissa de 7 bits. Qual é o erro relativo?

8.22 Os valores numéricos A e B são armazenados no computador como valores aproximados A' e B'. Desconsiderando qualquer truncamento ou erro de arredondamento, mostre que o erro relativo do produto é, aproximadamente, a soma dos erros relativos dos fatores.

8.23 Sendo $A = 1{,}427$, determine qual é o erro relativo, se A for truncado para 1,42 e se for arredondado para 1,43.

8.24 Um dos erros mais sérios em cálculos efetuados em computadores ocorre quando dois números quase iguais são subtraídos. Considere A = 0,22288 e B = 0,22211. Suponha que o computador trunque todo valor para quatro dígitos decimais. Portanto, A′ = 0,2228 e B′ = 0,2221.

a. Quais são os erros relativos em A' e B'?

b. Qual é o erro relativo em $C' = A' - B'$?

8.25 Mostre como as seguintes adições de ponto flutuante são efetuadas (onde as mantissas são truncadas para 4 dígitos decimais).
a. $0{,}5566 \times 10^3 + 0{,}7777 \times 10^3$ **b.** $0{,}3344 \times 10^2 + 0{,}8877 \times 10^{-1}$

8.26 Mostre como as seguintes subtrações de ponto flutuante são efetuadas (onde as mantissas são truncadas para 4 dígitos decimais).
a. $0{,}7744 \times 10^{-2} - 0{,}6666 \times 10^{-2}$ **b.** $0{,}8844 \times 10^{-2} - 0{,}2233 \times 10^{-2}$

8.27 Mostre como os seguintes cálculos de ponto flutuante são efetuados (onde as mantissas são truncadas para 4 dígitos decimais).
a. $(0{,}2255 \times 10^2) \times (0{,}1234 \times 10^1)$ **b.** $(0{,}8833 \times 10^3) \div (0{,}5555 \times 10^5)$

8.28 Expresse os seguintes números octais na notação hexadecimal:
a. 12 **b.** 5655 **c.** 2550276 **d.** 76545336 **e.** 3726755

8.29 Prove que todo número real com uma representação binária terminal (número finito de dígitos à direita da vírgula binária) tem também uma representação decimal terminal (número finito de dígitos à direita da vírgula decimal).

APÊNDICE 8A SISTEMAS DE NUMERAÇÃO

O sistema decimal

Na vida diária, usamos números representados em um sistema de numeração de base 10, com dígitos decimais (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), conhecido como sistema decimal. Considere o significado do número 83. Ele significa oito dezenas e três unidades:

$$83 = (8 \times 10) + 3$$

O número 4728 significa quatro milhares, sete centenas, duas dezenas e oito unidades:

$$4728 = (4 \times 1000) + (7 \times 100) + (2 \times 10) + 8$$

O sistema decimal é assim chamado por usar a *base* 10. Isso significa que cada dígito do número é multiplicado por 10 elevado à potência correspondente à posição do dígito:

$$83 = (8 \times 10^1) + (3 \times 10^0)$$

$$4728 = (4 \times 10^3) + (7 \times 10^2) + (2 \times 10^1) + (8 \times 10^0)$$

Valores fracionários são representados do mesmo modo:

$$472{,}83 = (4 \times 10^2) + (7 \times 10^1) + (2 \times 10^0) + (8 \times 10^{-1}) + (3 \times 10^{-2})$$

Em geral, para a representação decimal de $X = \{\ldots x_2x_1x_0 \ldots x_{-1}x_{-2}x_{-3}\ldots\}$, o valor de X é igual a:

$$X = \sum_i x_i 10^i$$

O sistema binário

No sistema decimal, são usados dez dígitos distintos para representar os números na base 10. No sistema binário, temos apenas dois dígitos, 1 e 0. Portanto, números no sistema binário são representados na base 2.

Para evitar confusão, algumas vezes usamos um número subscrito para indicar a base do sistema de numeração adotado. Por exemplo, 83_{10} e 4728_{10} são números representados na

notação decimal, ou seja, números decimais. Os dígitos 1 e 0, na notação binária, têm o mesmo significado que na notação decimal:

$$0_2 = 0_{10}$$

$$1_2 = 1_{10}$$

Para representar números maiores, assim como na notação decimal, cada dígito de um número binário tem um dado valor, dependendo de sua posição:

$$10_2 = (1 \times 2^1) + (0 \times 2^0) = 2_{10}$$

$$11_2 = (1 \times 2^1) + (1 \times 2^0) = 3_{10}$$

$$100_2 = (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4_{10}$$

e assim por diante. Valores fracionários são representados com potências negativas da base:

$$1001{,}101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9{,}625_{10}$$

Conversão entre números binários e decimais

A conversão de um número na notação binária para a notação decimal é simples. De fato, mostramos diversos exemplos na subseção anterior. Basta multiplicar cada dígito binário pela potência de 2 adequada e somar os resultados.

Para converter a notação decimal em notação binária, o número inteiro e a parte fracionária são tratados separadamente. Suponha que queremos converter um número inteiro decimal N para a forma binária. Se dividirmos N por 2, no sistema decimal, obtendo um quociente N_1 e um resto R_1, podemos escrever:

$$N = 2 \times N_1 + R_1 \qquad R_1 = 0 \text{ ou } 1$$

A seguir, dividimos o quociente N_1 por 2. Suponha que o novo quociente seja N_2 e o novo resto, R_2. Então:

$$N_1 = 2 \times N_2 + R_2 \qquad R_2 = 0 \text{ ou } 1$$

assim:

$$N = 2(2N_2 + R_2) + R_1 = 2^2N_2 + R_2 \times 2^1 \times \mathrm{R}_1 \times 2^0$$

Se, a seguir, tivermos:

$$N_2 = 2N_3 + R_3$$

então obteremos:

$$N = 2^3N_3 + R_3 \times 2^2 + R_2 \times 2^1 + R_1 \times 2^0$$

Como $N > N_1 > N_2$ ..., essa seqüência de divisões finalmente produzirá um quociente $N_k = 1$ (exceto para os números inteiros decimais 0 e 1, cuja notação binária é 0 e 1, respectivamente) e um resto R_k, que é igual a 0 ou 1. Então:

$$N = (1 \times 2^k) + (R_k \times 2^{k-1}) + \ldots + (R_3 \times 2^2) + (R_2 \times 2^1) + (\mathrm{R}_1 \times 2^0)$$

Ou seja, podemos converter da base 10 para a base 2 por meio de repetidas divisões por 2. O resto e o quociente final, 1, nos dão os dígitos binários de N, na ordem do menor para o maior dígito significativo. A Figura 8.27 mostra dois exemplos.

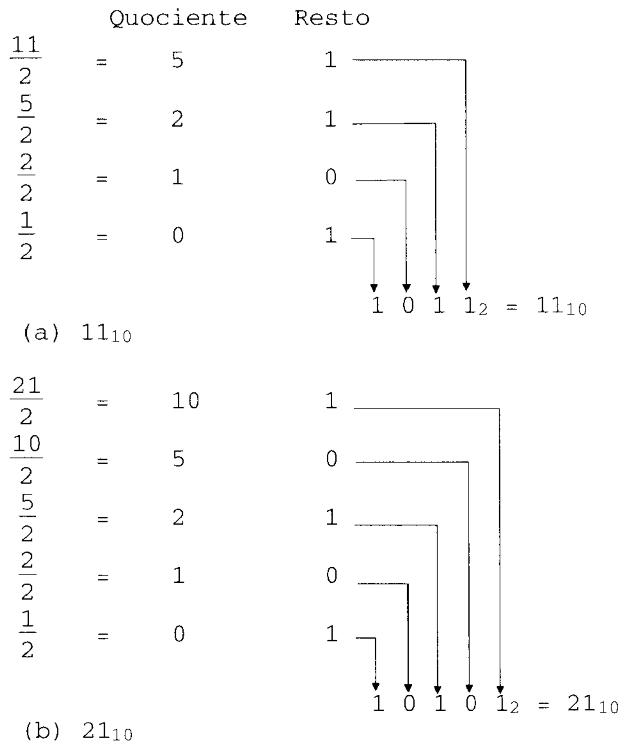


Figura 8.27 Exemplos de conversão de números inteiros da notação decimal para a notação binária.

A conversão da parte fracionária envolve repetidas multiplicações por dois, como mostra a Figura 8.28. A cada passo, a parte fracionária do número decimal é multiplicada por 2. O dígito à esquerda da vírgula decimal no produto será 0 ou 1 e contribuirá para a representação binária, começando pelo bit mais significativo. A parte fracionária do produto é usada como multiplicando no próximo passo. Para mostrar que isso funciona, consideramos uma fração decimal positiva $F < 1$. Podemos expressar F como:

$$F = \left(a_{-1} \times \frac{1}{2}\right) + \left(a_{-2} \times \frac{1}{2^2}\right) + \left(a_{-3} \times \frac{1}{2^3}\right) + \ldots$$

onde cada a_{-i} é 0 ou 1. Se multiplicarmos isso por 2, teremos:

$$2F = a_{-1} + \left(a_{-2} \times \frac{1}{2}\right) + \left(a_{-3} \times \frac{1}{2^2}\right) + \left(a_{-4} \times \frac{1}{2^3}\right) + \ldots$$

As partes inteiras dessas duas expressões devem ser iguais. Conseqüentemente, a parte inteira de $2F$, que deve ser 0 ou 1, uma vez que $0 < F < 1$, é simplesmente a_{-1}. Portanto: $2F = a_{-1} + F_1$, onde $0 < F_1 < 1$ e:

$$F_1 = \left(a_{-2} \times \frac{1}{2}\right) + \left(a_{-3} \times \frac{1}{2^2}\right) + \left(a_{-4} \times \frac{1}{2^3}\right) + \ldots$$

Para encontrar a_{-2} repetimos o mesmo processo, que não é necessariamente exato. Ou seja, uma fração decimal com um número finito de dígitos pode demandar uma fração binária com número infinito de dígitos. Nesses casos, o algoritmo de conversão normalmente é interrompido depois do número predefinido de passos, dependendo da precisão desejada.

Notação hexadecimal

Em virtude da natureza binária inerente dos componentes de um computador digital, todas as formas de dados são representadas, dentro do computador, por códigos binários. Vimos, anteriormente, exemplos de códigos binários para caracteres e de notação binária para números inteiros. Mais adiante, veremos exemplos do uso de códigos binários para outros tipos de dados. No entanto, embora o sistema binário seja conveniente para computadores, é excessivamente ineficiente para seres humanos. Por isso, a maioria dos profissionais de computação que passam grande parte do tempo trabalhando com dados manipulados no computador prefere uma notação mais compacta.

Que notação usar? Uma possibilidade seria a notação decimal. Essa notação certamente é mais compacta que a notação binária, mas é desconfortável devido à tediosa conversão entre a base 2 e a base 10.

```
Produto                Parte inteira  ,1 1 0 0 1 1
0,81 × 2 = 1,62                    1
0,62 × 2 = 1,24                    1
0,24 × 2 = 0,48                    0
0,48 × 2 = 0,96                    0
0,96 × 2 = 1,92                    1
0,92 × 2 = 1,84                    1
```

(a) $0{,}81_{10} = 0{,}110011_2$ (aproximado)

```
                                  ,0 1
0,25 × 2 = 0,5                     0
0,5  × 2 = 1,0                     1
```

(b) $0{,}25_{10} = 0{,}01_2$ (exato)

Figura 8.28 Exemplos de conversão de números fracionários da notação decimal para a notação binária.

Em vez disso, é adotada uma notação conhecida como hexadecimal. Os dígitos binários são agrupados em conjuntos de quatro. A cada combinação possível de quatro dígitos binários é atribuído um símbolo, como a seguir:

0000 = 0	1000 = 8
0001 = 1	1001 = 9
0010 = 2	1010 = A
0011 = 3	1011 = B
0100 = 4	1100 = C
0101 = 5	1101 = D
0110 = 6	1110 = E
0111 = 7	1111 = F

Por serem usados 16 símbolos, a notação é chamada hexadecimal e esses 16 símbolos são os dígitos hexadecimais.

Uma seqüência de dígitos hexadecimais pode ser vista como uma representação de um número inteiro na base 16. Portanto,

$$1A_{16} = (1_{16} \times 16^1) + (A_{16} \times 16^0)$$
$$= (1_{10} \times 16^1) + (10_{10} \times 16^0) = 26$$

A notação hexadecimal é usada não apenas para representar números inteiros. Ela também é usada como uma notação concisa para representar qualquer seqüência de dígitos binários, mesmo que representem texto, números ou algum outro tipo de dado. As razões para se usar notação hexadecimal são as seguintes:

1. É mais compacta que a notação binária.
2. Na maioria dos computadores, os dados binários têm um tamanho que é múltiplo de 4 bits e, portanto, múltiplo de um dígito hexadecimal.
3. É extremamente fácil converter entre as notações binária e hexadecimal.

Como um exemplo desse último ponto, considere a seqüência de bits 110111100001. Isso é equivalente a:

```
1101  1110  0001 = DE1₁₆
  D     E     1
```

Esse processo é realizado tão naturalmente que um programador experiente pode converter representações visuais de dados binários para seus equivalentes hexadecimais mentalmente, sem precisar escrever. É bem provável que você nunca necessite dessa habilidade em particular. Entretanto, essa discussão foi incluída neste texto porque a notação hexadecimal é muito comum e você fatalmente encontrará algum texto em que ela é usada.

capítulo

9 CONJUNTO DE INSTRUÇÕES: CARACTERÍSTICAS E FUNÇÕES

Mapeamento de endereços big-endian

Endereço de byte								
	11	12	13	14				
00	00	01	02	03	04	05	06	07
	21	22	23	24	25	26	27	28
08	08	09	0A	0B	0C	0D	0E	0F
	31	32	33	34	'A'	'B'	'C'	'D'
10	10	11	12	13	14	15	16	17
	'E'	'F'	'G'		51	52		
18	18	19	1A	1B	1C	1D	1E	1F
	61	62	63	64				
20	20	21	22	23				

- Os elementos essenciais de uma instrução de computador são o código de operação, que especifica a operação a ser realizada, as referências aos operandos de origem e de destino, que especificam os endereços dos dados de entrada e de saída da operação, e o endereço da próxima instrução, que normalmente é implícito.
- As operações podem ser classificadas nas seguintes categorias gerais: operações lógicas e aritméticas, operações de movimentação de dados entre dois registradores, entre registrador e memória ou entre duas posições de memória, operações de E/S e operações de controle.
- As referências a operandos especificam um registrador ou uma posição de memória. Os tipos de dados podem ser endereços, números, caracteres ou valores lógicos.
- Uma característica comum à arquitetura de diversos processadores é o uso de uma pilha, que pode ou não ser visível para o programador. A pilha pode ser usada para gerenciar chamadas e retornos de procedimentos e pode constituir uma forma alternativa de endereçamento à memória. As operações básicas sobre pilhas são PUSH (EMPILHAR) e POP (DESEMPILHAR), além de operações que manipulam uma ou duas posições localizadas no topo da pilha. Tipicamente, uma pilha é implementada de modo que cresça no sentido de endereços mais altos para endereços mais baixos.
- Um processador pode manipular dados com ordenação de bytes big-endian, little-endian e bi-endian. Se um valor numérico de múltiplos bytes for armazenado na memória com o byte mais significativo no endereço mais baixo, então se dirá que ele usa uma ordenação de bytes big-endian; se for armazenado com o byte mais significativo no endereço mais alto, ele usará uma ordenação de bytes little-endian. Alguns processadores podem manipular dados armazenados na memória das duas formas.

A maioria dos aspectos discutidos neste livro não é imediatamente aparente para o usuário ou o programador do computador. Quando um programador usa uma linguagem de alto nível, como Pascal ou Ada, muito pouco da arquitetura da máquina subjacente é visível.

O conjunto de instruções da máquina constitui o limite em que o projetista e o programador de computadores enxergam uma mesma máquina. Do ponto de vista do projetista, o conjunto de instruções de máquina fornece os requisitos funcionais para a CPU: implementar uma CPU é uma tarefa que envolve, em grande parte, implementar o conjunto de instruções de máquina. Por outro lado, o usuário que decidir programar em linguagem de máquina (na verdade, em linguagem de montagem, discutida na Seção 9.6) deve ter conhecimento sobre o conjunto de registradores da CPU, a estrutura de memória, os tipos de dados disponíveis diretamente na máquina e o funcionamento da ULA.

Da descrição do conjunto de instruções de máquina a uma compreensão sobre o funcionamento da CPU, há um longo caminho. Assim, este capítulo e o próximo enfocam as instruções de máquina e, em seguida, iniciamos o estudo da estrutura e do funcionamento da CPU.

9.1 CARACTERÍSTICAS DE INSTRUÇÕES DE MÁQUINA

A operação de uma CPU é determinada pelas instruções que ela executa, conhecidas como *instruções de máquina* ou *instruções do computador*. A coleção de diferentes instruções que a CPU é capaz de executar é conhecida como *conjunto de instruções* da CPU.

Elementos de instruções de máquina

Cada instrução deve conter toda a informação necessária para que a CPU possa executá-la. A Figura 9.1, que repete a Figura 3.6, mostra os passos envolvidos na execução de instruções, definindo os elementos de instruções de máquina:

- **Código de operação:** especifica a operação a ser efetuada (por exemplo, ADD, E/S). A operação é especificada por um código binário, conhecido como código de operação.
- **Referência a operando fonte:** a operação pode envolver um ou mais operandos fonte, ou seja, operandos que constituem dados de entrada para a operação.
- **Referência a operando de destino:** a operação pode produzir um resultado.
- **Endereço da próxima instrução:** indica onde a CPU deve buscar a próxima instrução, depois que a execução da instrução corrente for completada.

A próxima instrução a ser buscada pode estar localizada na memória principal ou, no caso de um sistema com memória virtual, tanto na memória principal quanto na memória secundária (disco). Na maioria dos casos, a próxima instrução é a que segue imediatamente a instrução corrente. Nesses casos, a instrução não inclui uma referência explícita para a próxima instrução. Quando isso é necessário, a instrução deve fornecer um endereço de memória principal ou de memória virtual. O modo como esse endereço é fornecido será discutido no Capítulo 10.

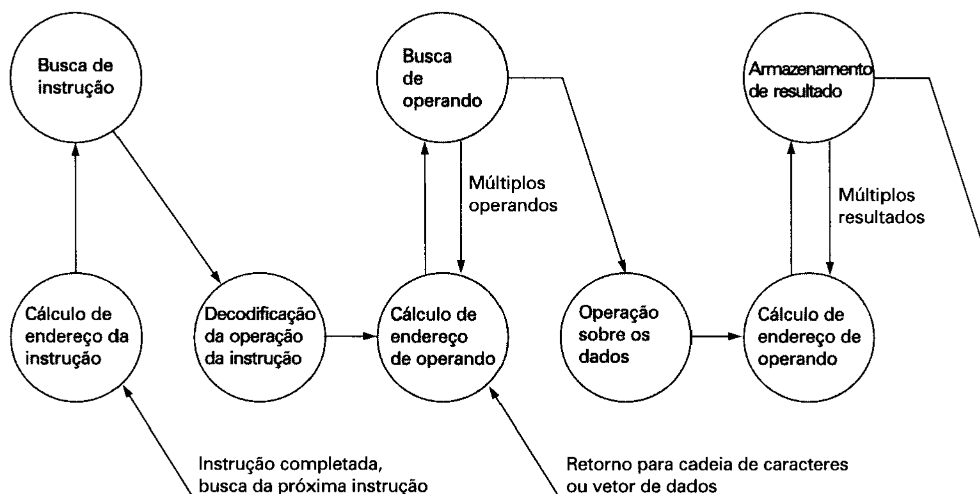


Figura 9.1 Diagrama de estados do ciclo de instruções.

Os operandos fonte e de destino podem estar localizados em uma das seguintes áreas:

- **Memória principal ou virtual:** assim como na referência para a próxima instrução, deve ser fornecido um endereço, que pode ser na memória principal ou na memória virtual.
- **Registrador da CPU:** com raras exceções, a CPU contém um ou mais registradores, que podem ser referenciados pelas instruções de máquina. Se existir apenas um único registrador, a referência a ele poderá ser implícita. Se existirem vários registradores, então, cada registrador será designado por um número distinto, e a instrução deverá conter o número do registrador desejado.
- **Dispositivo de E/S:** a instrução deve especificar um módulo de E/S e um dispositivo para a operação. Se for usada a E/S mapeada na memória, essa informação consistirá apenas em um endereço na memória principal ou na memória virtual.

Representação de instruções

Internamente, cada instrução de um computador é representada como uma seqüência de bits. Uma instrução é dividida em campos, correspondentes aos elementos da instrução. Um exemplo simples de formato de instrução é mostrado na Figura 9.2. Outro exemplo é o formato de instrução do IAS, apresentado na Figura 2.2. Na maioria dos conjuntos de instruções, é usado mais de um formato de instrução. Durante a execução, uma instrução é lida em um registrador de instruções (IR) da CPU. A CPU deve ser capaz de extrair os dados dos vários campos da instrução e efetuar a operação requerida.

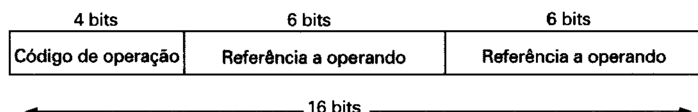


Figura 9.2 Formato de instrução simples.

É difícil para o programador (e também para o leitor de livros-texto) lidar com representações binárias de instruções de máquina. Por isso, tornou-se prática comum usar uma *representação simbólica* para instruções de máquina. Um exemplo disso foi apresentado na Tabela 2.1, para o conjunto de instruções do IAS.

Os códigos de operação são representados por abreviações*, chamadas *mnemônicos*, que indicam a operação a ser efetuada. Alguns exemplos comuns são:

ADD	Adição
SUB	Subtração
MPY	Multiplicação
DIV	Divisão
LOAD	Carregar dados da memória
STOR	Armazenar dados na memória

Os operandos são também representados de maneira simbólica. Por exemplo, a instrução

```
ADD R, Y
```

pode significar adicionar o valor contido na posição *Y* com o conteúdo do registrador *R*. Nesse exemplo, *Y* é um endereço de uma posição de memória e *R* indica um registrador particular. Note que a operação é feita sobre o conteúdo da posição de memória, e não sobre seu endereço.

Portanto, é possível escrever um programa em linguagem de máquina de maneira simbólica. Cada código de operação simbólico tem uma representação binária correspondente, e o programador especifica o endereço de cada operando simbólico. Por exemplo, o programador pode começar com a seguinte lista de definições:

```
X = 513
Y = 514
```

e assim por diante. Essa entrada simbólica pode ser convertida, por meio de um programa bastante simples, em códigos de operação e referências a operandos na forma binária, resultando em instruções de máquina binárias.

Hoje, é muito raro programar em linguagem de máquina. A maioria dos programas é escrita em linguagem de alto nível ou, em alguns casos, em linguagem de montagem, que será discutida no final deste capítulo. No entanto, a linguagem de máquina simbólica permanece como uma ferramenta útil para descrever instruções de máquina, sendo usada a seguir para esse propósito.

Tipos de instrução

Considere uma instrução em uma linguagem de alto nível, tal como BASIC ou FORTRAN. Por exemplo:

```
X = X + Y
```

Esse comando instrui o computador a adicionar o valor armazenado em Y ao valor armazenado em X e colocar o resultado em X. Como isso pode ser feito usando instruções de

* N.R.T.: Normalmente, os mnemônicos das instruções são abreviações das palavras em inglês, como, por exemplo, MPY (*multiply*) e STOR (*store*).

máquina? Suponha que as variáveis X e Y correspondam às posições de memória de endereços 513 e 514. Se considerarmos um conjunto simples de instruções de máquina, esse comando pode ser implementado com três instruções:

1. Carregar um registrador com o conteúdo da posição de memória 513.
2. Adicionar o conteúdo da posição de memória 514 ao registrador.
3. Armazenar o conteúdo do registrador na posição de memória 513.

Como se pode observar, uma única instrução em linguagem de alto nível pode requerer várias instruções de máquina. Isso é típico do relacionamento entre uma linguagem de alto nível e uma linguagem de máquina. Em uma linguagem de alto nível, as operações são expressas de uma maneira algébrica concisa, usando variáveis. Em uma linguagem de máquina, as operações são expressas de maneira mais básica, envolvendo a movimentação de dados de e para registradores.

Tendo como base esse exemplo simples, consideramos os tipos de instruções que devem ser incluídos em um computador. Um computador deve ter um conjunto de instruções que permita ao usuário formular qualquer tarefa de processamento de dados. Outra maneira de determinar esse conjunto de instruções é considerar os comandos disponíveis em uma linguagem de programação de alto nível. Qualquer programa em uma linguagem de alto nível deve ser traduzido para uma linguagem de máquina, para que possa ser executado. Portanto, o conjunto de instruções de máquina deve ser suficiente para expressar qualquer comando de uma linguagem de alto nível. Com isso em mente, podemos, então, catalogar os tipos de instruções de máquina como a seguir:

- **Processamento de dados:** instruções aritméticas e lógicas
- **Armazenamento de dados:** instruções de memória
- **Movimentação de dados:** instruções de E/S
- **Controle:** instruções de teste e de desvio

Instruções aritméticas fornecem a capacidade computacional para processamento de dados numéricos. *Instruções lógicas* (booleanas) operam sobre bits de uma palavra, como bits e não como números; oferecem, portanto, a capacidade para processar qualquer outro tipo de dado que o usuário possa desejar empregar. Essas operações são efetuadas, primariamente, em dados armazenados em registradores da CPU. Por isso, devem existir *instruções de memória* para mover dados entre a memória e os registradores. *Instruções de E/S* são necessárias para transferir programas e dados para a memória e para transferir resultados da computação de volta para o usuário. *Instruções de teste* são usadas para testar o valor de uma palavra de dados ou o estado de uma computação. *Instruções de desvio* são utilizadas para desviar a execução do programa para uma nova instrução, possivelmente dependendo do resultado de um teste.

Esses vários tipos de instrução são examinados detalhadamente neste capítulo.

Número de endereços

Uma das maneiras tradicionais de descrever uma arquitetura é em termos do número de endereços contidos em cada instrução. Esse aspecto tornou-se menos significativo com a crescente complexidade de projeto de CPU. Entretanto, é útil considerar e analisar essa distinção entre instruções com diferentes números de endereços.

Qual é o número máximo de endereços necessários em uma instrução? Evidentemente, instruções aritméticas e lógicas requerem maior número de operandos. Quase todas as operações aritméticas e lógicas são unárias (um operando) ou binárias (dois operandos). Portanto, precisamos, no máximo, de dois endereços para referenciar operandos. Como o resultado da operação deve ser armazenado, isso sugere que é necessário um terceiro endereço. Finalmente, depois de concluída a execução de uma instrução, a próxima instrução deve ser buscada, sendo necessário conhecer seu endereço.

Essa linha de raciocínio sugere que poderia ser necessário ter instruções com quatro endereços: dois para operandos, um para o resultado e o endereço da próxima instrução. Na prática, instruções com quatro endereços são extremamente raras. A maioria das instruções tem um, dois ou três endereços de operando, sendo implícito o endereço da próxima instrução (contido no contador de programa).

A Figura 9.3 compara instruções típicas de um, dois e três endereços, que podem ser usadas para computar o comando $Y = (A - B) \div (C + D \times E)$. Com instruções de três endereços, cada instrução especifica dois endereços de operandos e um endereço para o resultado. Como podemos querer não alterar o valor de qualquer posição de memória, uma área de memória temporária, T, é usada para armazenar resultados intermediários. Note que a implementação do comando requer quatro instruções e que a expressão original tem cinco operandos.

Instrução		Comentário
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Instruções com três endereços

Instrução		Comentário
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Instruções com dois endereços

Instrução		Comentário
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) Instruções com um endereço

Figura 9.3 Programas para executar o comando $Y = (A - B) \div (C + D \times E)$.

Formatos de instrução com três endereços não são muito comuns, porque resultam em instruções de tamanho relativamente grande, devido ao espaço necessário para manter os três endereços. No caso de instruções com dois endereços e das operações binárias, um dos endereços referencia tanto um operando quanto o resultado. Por exemplo, a instrução SUB Y, B calcula o valor Y – B e armazena o resultado em Y. O uso do formato de instrução com dois endereços reduz o tamanho das instruções, mas apresenta algumas desvantagens. Para evitar que se altere o valor de um operando, é usada uma instrução MOVE, para mover um dos operandos para a posição de resultado ou para uma posição temporária, antes de a operação ser efetuada. O número de instruções requeridas para implementar o exemplo anterior aumentaria, nesse caso, para seis instruções.

Instruções de apenas um endereço são ainda mais simples. Nesse caso, um segundo endereço deve ser implícito. Esse tipo de instrução era comum nas primeiras máquinas, onde o endereço subentendido é um registrador da CPU, conhecido como *acumulador* (AC). O acumulador contém um dos operandos e é usado para armazenar o resultado. Nesse caso, são necessárias oito instruções para implementar nosso exemplo.

É possível, ainda, usar um formato de instrução com zero endereço, para alguns tipos de instrução. Esse formato se aplica a uma organização de memória especial, denominada *pilha*. Uma pilha consiste em um conjunto de posições de memória, que são manipuladas de maneira que cada leitura efetuada sobre a pilha recupere o último dado nela armazenado (topo da pilha), retirando-o da pilha*. A área de memória reservada para a pilha inicia a partir de um endereço fixo na memória e, usualmente, pelo menos os dois elementos no topo da pilha são armazenados em registradores da CPU. Instruções de zero endereços referenciam os dois elementos no topo da pilha. Pilhas são descritas no Apêndice 9A. Suas aplicações são exploradas mais adiante, neste e no próximo capítulo.

Tabela 9.1 Utilização de endereços em instruções (exceto instruções de desvio)

Número de endereços	Representação simbólica	Interpretação
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T–1) OP T

AC = acumulador
T = topo da pilha
A, B, C = registrador ou posição de memória

A Tabela 9.1 apresenta um resumo da interpretação de instruções com zero, um, dois ou três endereços. Em cada caso, supomos que o endereço da próxima instrução a ser executada seja implícito e que a operação requeira dois operandos como entrada e um operando como resultado.

O número de endereços por instrução constitui uma decisão de projeto importante. Poucos endereços por instrução resultam em instruções de menor extensão e mais primitivas, que requerem uma CPU menos complexa. Por outro lado, o número de instruções por programa é maior, o que, em geral, resulta em maior tempo de execução e em programas mais complexos. Além disso, existe outro aspecto importante em relação ao qual instruções de um ou múltiplos endereços se contrapõem. Com instruções de um endereço, o programador geralmente tem disponível apenas um registrador de propósito geral, o acumulador. Com instruções de múltiplos endereços, é comum haver múltiplos registradores de propósito geral. Isso possibilita que algumas operações sejam efetuadas apenas sobre registradores. Como referências a registradores são mais rápidas do que referências à memória, a execução dessas instruções é mais

* N.R.T.: Essa organização é conhecida como LIFO (*last-in first-out*), ou seja, o primeiro a entrar é o último a sair.

rápida. Em razão da flexibilidade e da capacidade para usar múltiplos registradores, a maioria das máquinas modernas emprega instruções de dois ou de três endereços.

As questões de projeto envolvidas na escolha do número de endereços por instrução são complicadas ainda por outros fatores. Um deles consiste em decidir se um endereço se refere a uma posição de memória ou a um registrador. Como o número de registradores é menor que o número de posições de memória, um número menor de bits é requerido para endereçar um registrador. Além disso, como veremos no próximo capítulo, uma máquina pode oferecer uma variedade de modos de endereçamento, requerendo um ou mais bits para especificar o modo de endereçamento. Como resultado, a maioria dos projetos de CPUs envolve uma variedade de formatos de instrução.

Projeto do conjunto de instruções

Um dos aspectos mais interessantes e mais analisados do projeto de computadores é o projeto do conjunto de instruções. O projeto do conjunto de instruções é muito complexo, pois afeta diversos aspectos do sistema. Ele define muitas das funções desempenhadas pela CPU e, portanto, tem efeito significativo sobre a implementação da CPU. Como o conjunto de instruções constitui o meio pelo qual o programador pode controlar a CPU, ao projetar um conjunto de instruções é preciso considerar as necessidades do programador.

Você poderá se surpreender ao saber que algumas das questões mais fundamentais relativas ao projeto de conjuntos de instruções permanecem ainda em discussão. De fato, nos últimos anos, tem crescido a discordância a respeito dessas questões. Algumas das questões mais importantes são as seguintes:

- **Repertório de operações:** quantas e quais são as operações que devem ser fornecidas e quão complexas elas podem ser.
- **Tipos de dados:** quais os tipos de dados sobre os quais as operações são efetuadas.
- **Formatos de instrução:** qual o tamanho das instruções (em bits), o número de endereços por instrução, o tamanho dos vários campos etc.
- **Registradores:** qual o número de registradores da CPU que podem ser usados pelas instruções e qual o propósito de cada um.
- **Endereçamento:** de que modo (ou modos) o endereço de um operando pode ser especificado.

Essas questões são altamente inter-relacionadas e devem ser consideradas em conjunto ao se projetar um conjunto de instruções. É claro que, neste livro, temos de considerá-las em alguma seqüência, mas procuramos sempre mostrar o relacionamento entre elas.

Em virtude da importância do tópico, a maioria das seções da Parte III é dedicada ao projeto de conjuntos de instruções. Depois dessa primeira seção de visão geral, este capítulo discutirá os tipos de dados e o repertório de operações. O Capítulo 10 examinará os possíveis modos de endereçamento (o que inclui considerações sobre registradores) e os formatos de instruções. O Capítulo 12 abordará computadores com um conjunto reduzido de instruções (*reduced instruction set computer* — RISC). A descrição da arquitetura RISC suscita diversas questões relativas a decisões de projeto de conjuntos de instruções de computadores comerciais contemporâneos. Um exemplo de máquina RISC é o PowerPC, usado como um dos exemplos neste e no próximo capítulo. A importância do projeto do PowerPC no contexto de máquinas RISC será discutida no Capítulo 12.

9.2 TIPOS DE OPERANDOS

Instruções de máquina operam sobre dados. As classes de dados mais importantes são:

- Endereços
- Números
- Caracteres
- Dados lógicos

Ao discutir os modos de endereçamento no Capítulo 10, veremos que os endereços são, de fato, uma forma de dado. Em muitos casos, é necessário efetuar cálculos sobre o valor do campo de endereço de um operando de uma instrução para determinar o endereço de memória principal ou virtual correspondente. Nesse contexto, os endereços podem ser considerados números inteiros sem sinal.

Outros tipos de dados comuns são números, caracteres e dados lógicos; cada um será discutido brevemente nesta seção. Algumas máquinas definem, além desses, alguns tipos de dados especiais ou estruturas de dados. Por exemplo, podem fornecer operadores que operam diretamente sobre listas ou seqüências de caracteres.

Números

Toda linguagem de máquina inclui tipos de dados numéricos. Mesmo o processamento de dados não numéricos requer o uso de números, como valor de contadores, tamanho de campos etc. Uma distinção importante entre números usados na matemática usual e números armazenados em um computador é que esses últimos são limitados em dois sentidos. Primeiro, existe um limite para a magnitude de números representáveis em uma máquina; segundo, no caso de números de ponto flutuante, há um limite para sua precisão. Um programador precisa, portanto, entender os efeitos do arredondamento, *overflow* e *underflow*.

Três tipos de dados numéricos são comuns em computadores:

- Número inteiro ou de ponto fixo
- Número de ponto flutuante
- Número decimal

Os dois primeiros foram examinados detalhadamente no Capítulo 8. Resta-nos dizer algumas poucas palavras sobre números decimais.

Embora toda operação interna de um computador seja, por natureza, binária, usuários do sistema lidam com números decimais. Portanto, números decimais devem ser convertidos para números binários, na entrada, e números binários devem ser convertidos para números decimais, na saída. Para aplicações com grande quantidade de E/S e relativamente pouca computação, é preferível armazenar e operar sobre números na forma decimal. A representação mais comumente usada para esse propósito é a representação de número decimal empacotado em unidades de 4 bits (essa representação é conhecida como BCD — *binary coded decimal*).

Nessa notação, cada dígito decimal é representado por um código de 4 bits, de maneira óbvia. Portanto, 0 = 0000, 1 = 0001, ..., 8 = 1000, 9 = 1001. Note que esse código é um tanto ineficiente, pois são usados apenas 10 dos 16 valores possíveis que podem ser armazenados em 4 bits. Um número é representado como uma seqüência desses códigos de 4 bits, normalmente com tamanho múltiplo de 8 bits. Por exemplo, o código para 246 é 0000001001000110.

Esse código é, claramente, menos compacto do que uma representação binária direta, mas evita o custo de conversão. Números negativos podem ser representados incluindo um dígito de sinal de 4 bits, seja na extremidade esquerda seja na extremidade direita da seqüência de dígitos decimais empacotados. Por exemplo, o código 1111 pode ser usado para o sinal negativo.

Muitas máquinas oferecem instruções aritméticas para operar diretamente sobre números decimais representados dessa maneira. Os algoritmos usados para essas operações são semelhantes aos descritos na Seção 8.3 e devem levar em conta as operações de 'vai-um' decimal.

Caracteres

Uma forma comum de dado é o texto ou a seqüência de caracteres. Embora dados textuais sejam convenientes para o ser humano, eles não podem ser armazenados ou transmitidos facilmente, na forma de caracteres, por sistemas de processamento de dados ou de comunicação, uma vez que esses sistemas são projetados para manipular dados binários. Portanto, foram criados diversos códigos onde caracteres são representados por seqüências de bits. O exemplo mais antigo desse tipo de código talvez seja o código Morse. Hoje em dia, o código de caracteres mais usado é o Alfabeto de Referência Internacional (*International Reference Alphabet* — IRA), conhecido como código ASCII (*American Standard Code for Information Interchange*) (Tabela 6.1). Cada caractere desse código é representado por um padrão distinto de 7 bits; assim, 128 caracteres diferentes podem ser representados. Esse número é maior que o necessário para representar caracteres visíveis, sendo alguns padrões de bits usados para representar caracteres de *controle*. Alguns desses caracteres de controle são usados para controlar a impressão de caracteres em uma página. Outros são usados para controlar procedimentos de comunicação. Caracteres ASCII são quase sempre armazenados e transmitidos usando 8 bits por caractere. O oitavo bit pode ser sempre 0 ou pode ser usado como bit de paridade, para detecção de erro. Nesse caso, o valor atribuído a esse bit é tal que o número total de dígitos binários iguais a 1, em cada conjunto de 8 bits, seja sempre ímpar (paridade ímpar) ou seja sempre par (paridade par).

Note, pela Tabela 6.1, que, em um padrão de bits da forma 011XXXX, no código ASCII, os dígitos 0 a 9 são representados por seus valores binários 0000 a 1001, nos 4 bits mais à direita. Esse é o mesmo código usado para os dígitos 0 a 9 na representação de número decimal empacotado descrita anteriormente, onde cada dígito decimal é representado com 4 bits. Isso facilita a conversão entre o código ASCII de 7 bits e a representação de número decimal empacotado de 4 bits.

Outro código usado para codificar caracteres é o EBCDIC (*Extended Binary Coded Decimal Interchange Code*). O EBCDIC é um código de 8 bits, usado nas máquinas IBM S/370. Assim como o código ASCII, o EBCDIC é compatível com a representação decimal empacotada. No EBCDIC, os dígitos 0 a 9 são representados pelos códigos 11110000 a 11111001.

Dados lógicos

Normalmente, cada palavra ou unidade endereçável (byte, meia palavra etc.) é tratada como uma unidade única de dado. Entretanto, algumas vezes é útil considerar uma unidade de *n* bits como composta de *n* itens de dado, de um 1 bit cada, com valor 0 ou 1. Quando um dado é visto dessa maneira, ele é considerado um dado *lógico*.

Essa visão orientada a bits apresenta duas vantagens. A primeira é a economia de memória obtida quando queremos armazenar vetores de dados booleanos, onde cada dado apenas pode ter valor 1 (verdadeiro) ou 0 (falso). A segunda é possibilitar a manipulação de bits de um item de dado, o que pode ser requerido em determinadas situações. Por exemplo, a implementação de operações de ponto flutuante por software requer a capacidade para deslocar bits significativos de um operando, em determinadas operações. Outro exemplo é a conversão de código ASCII para representação decimal empacotada, em que precisamos extrair os 4 bits mais à direita do byte.

Note, pelos exemplos anteriores, que um mesmo dado pode ser tratado algumas vezes como um dado lógico e outras vezes como um dado numérico ou textual. O 'tipo' de um dado é determinado pela operação efetuada sobre ele. Isso normalmente não ocorre em linguagens de alto nível (por exemplo, em Pascal), mas é quase sempre o caso em linguagens de máquina.

9.3 TIPOS DE DADOS DO PENTIUM II E DO PowerPC

Tipos de dados do Pentium II

O Pentium II pode lidar com tipos de dados com tamanho de 8 bits (byte), 16 bits (palavra), 32 bits (palavra dupla) e 64 bits (palavra quádrupla). Para possibilitar máxima flexibilidade na manipulação de estruturas de dados e na utilização eficiente de memória, as palavras não precisam ser alinhadas em endereços de número par, as palavras duplas não precisam ser alinhadas em endereços divisíveis por 4 e as palavras quádruplas não precisam ser alinhadas em endereços divisíveis por 8. No entanto, quando um dado é transferido por meio de um barramento de 32 bits, a transferência se dá em unidades de palavra dupla, começando em endereços divisíveis por 4. O processador converte uma requisição de transferência de um valor não alinhado em uma seqüência de requisições adequadas para transferência por meio do barramento. Assim como todas as máquinas Intel 80x86, o Pentium II adota a disposição de byte little-endian; isto é, o byte menos significativo é armazenado no endereço mais baixo (veja o Apêndice 9B para uma discussão sobre disposição de bytes dos dados).

Os tipos byte, palavra, palavra dupla e palavra quádrupla são conhecidos como tipos de dados gerais. Além desses tipos de dados, o Pentium II oferece suporte a vetores de dados, para alguns tipos de dados particulares, e operações específicas, para operar sobre esses vetores. Como mostra a Figura 9.4, as instruções de ponto flutuante operaram sobre números inteiros, inteiros na representação decimal empacotada e sobre números de ponto flutuante. Os números inteiros usam a representação em complemento de dois, com tamanhos de 16, 32 ou 64 bits. Números inteiros na representação decimal empacotada são armazenados no formato sinal-magnitude, com 18 dígitos, no intervalo 0 a 9. As três representações de números de ponto flutuante usadas no Pentium II estão de acordo com o padrão IEEE 754.

Tabela 9.2 Tipos de dados do Pentium II

Tipos de dados	Descrição
Gerais	Endereço de byte, palavra (16 bits), palavra dupla (32 bits) ou palavra quádrupla (64 bits), com conteúdo binário arbitrário.
Número inteiro	Valor binário sem sinal, contido em um byte, palavra ou palavra dupla, usando representação em complemento de dois.
Número ordinal	Número inteiro sem sinal, contido em um byte, palavra ou palavra dupla.
Decimal desempacotado (*Unpacked BCD*)	Representação de um dígito BCD, com valor entre 0 e 9, com um dígito em cada byte.
Decimal empacotado (*Packed BCD*)	Representação de dois dígitos BCD empacotados em um byte; valor entre 0 e 99.
Apontador (*Near Pointer*)	Endereço efetivo de 32 bits, que representa um endereço relativo ao início de um segmento. Usado para todos os apontadores para posições em uma memória não-segmentada e para referências dentro de um segmento, em uma memória segmentada.
Campo de bits	Seqüência contígua de bits, onde cada posição de bit é considerada uma unidade independente. Uma seqüência de bits pode começar em qualquer posição de bit de qualquer byte e pode conter até $2^{32}-1$ bits.
Seqüência de bytes	Seqüência contígua de bytes, palavras ou palavras duplas que contém de 0 a $2^{32}-1$ bytes.
Ponto flutuante	Veja Figura 9.4.

Tipos de dados do PowerPC

O PowerPC pode lidar com tipos de dados com tamanhos de 8 bits (byte), 16 bits (meia palavra), 32 bits (palavra) e 64 bits (palavra dupla). Algumas instruções requerem que os operandos de memória estejam alinhados em posições-limite de 32 bits. Entretanto, de modo geral, esse alinhamento não é requerido. Uma característica interessante do PowerPC é que ele pode usar tanto a disposição de bytes big-endian quanto a little-endian; ou seja, o byte menos significativo pode ser armazenado no endereço mais baixo ou no endereço mais alto (veja o Apêndice 9B para uma discussão sobre disposição de bytes dos dados).

Os tipos byte, meia palavra, palavra e palavra dupla são tipos de dados gerais. O processador interpreta o conteúdo de um determinado item de dado de acordo com a instrução. O processador de ponto fixo reconhece os seguintes tipos de dados:

- **Byte sem sinal:** pode ser usado em operações aritméticas de número inteiro ou em operações lógicas. É carregado da memória em um registrador de propósito geral que estende seu valor com zeros à esquerda, até o tamanho total do registrador.
- **Meia palavra sem sinal:** análoga ao byte sem sinal, mas com 16 bits.

- **Meia palavra com sinal:** usada em operações aritméticas, é carregada da memória em um registrador de propósito geral, que estende seu valor pela replicação do bit de sinal em todas as posições vagas à esquerda, até o tamanho total do registrador.
- **Palavra sem sinal:** usada em operações lógicas e como endereço.
- **Palavra com sinal:** usada em operações aritméticas.
- **Palavra dupla sem sinal:** usada como endereço.
- **Seqüência de bytes:** com tamanho de 0 a 128 bytes.

Além desses tipos de dados, o PowerPC oferece suporte a números de ponto flutuante de precisão simples e dupla, como definidos no padrão IEEE 754.

Formato de dados	Faixa de valores	Precisão	Byte mais significativo (7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0 | 7 0) Byte de endereço mais alto
Inteiro de palavra	10^4	16 bits	Complemento de dois (15 … 0)
Inteiro de palavra dupla	10^9	32 bits	Complemento de dois (31 … 0)
Inteiro de palavra dupla	10^{18}	64 bits	Complemento de dois (63 … 0)
BCD empacotado	10^{18}	18 Digits	s (79) | X (71) | d_{17} d_{16} d_{15} d_{14} d_{13} d_{12} d_{11} d_{10} d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 (0)
Ponto flutuante de precisão simples	$10^{\pm 38}$	24 bits	s (31) | Expoente polarizado (22) | Mantissa (0)
Ponto flutuante de precisão dupla	$10^{\pm 308}$	53 bits	s (63) | Expoente polarizado (51) | Mantissa (0)
Ponto flutuante de precisão estendida	$10^{\pm 4932}$	64 bits	s (79) | Expoente polarizado | I (63) D | Mantissa (0)

s = bit de sinal (0 = positivo; 1 = negativo)

d_n = dígito decimal (dois por byte)

X = bits sem significado; ignorados quando o valor é carregado; iguais a zero quando o valor é armazenado

D = posição do ponto binário implícito

I = bit inteiro da mantissa; explícito para valor armazenado em registrador; implícito para valor de precisão simples ou dupla armazenado na memória

Figura 9.4 Formatos de dados numéricos do Pentium II.

9.4 TIPOS DE OPERAÇÕES

O número de códigos de operação distintos varia muito de máquina para máquina. Entretanto, o mesmo conjunto de classes de operações é encontrado em todas as máquinas. Uma classificação típica dessas operações é descrita a seguir:

- Operações de transferência de dados
- Operações aritméticas
- Operações lógicas
- Operações de conversão
- Operações de E/S
- Operações de controle de sistema
- Operações de transferência de controle

A Tabela 9.3 (baseada em Hayes, 1988) enumera tipos de instrução comuns em cada classe. Esta seção apresenta uma breve descrição dos vários tipos de operações, juntamente com uma discussão sucinta sobre as ações tomadas pela CPU para executar cada tipo particular de operação (resumidas na Tabela 9.4). Esse último tópico será examinado mais detalhadamente no Capítulo 11.

Operações de transferência de dados

O tipo mais fundamental de instrução de máquina é a instrução de transferência de dados. Essa instrução deve especificar várias informações. Primeiramente, deve especificar os endereços dos operandos fonte e de destino da operação. Cada endereço pode indicar uma posição de memória, um registrador ou o topo da pilha. Em segundo lugar, deve indicar o tamanho dos dados a serem transferidos. Em terceiro, como em todas as demais instruções com operandos, deve especificar o modo de endereçamento de cada operando. Esse último aspecto será discutido no Capítulo 10.

A escolha das instruções de transferência de dados que devem ser incluídas em um conjunto de instruções mostra vários tipos de decisão que um projetista deve tomar. Por exemplo, o endereço (memória ou registrador) de um operando pode ser indicado tanto pelo código da operação quanto pela especificação do operando. A Tabela 9.5 mostra exemplos das instruções de transferências de dados mais comuns do IBM S/370. Note que existem variantes para cada quantidade de dados a serem transferidos (8, 16, 32 ou 64 bits). Além disso, há também diferentes instruções para a transferência de dados de registrador para registrador, de registrador para memória e de memória para registrador. Em contraposição, o VAX fornece uma instrução de transferência (MOV), com variantes para as diferentes quantidades de dados a serem movidos, mas especifica se um operando está em um registrador ou na memória como parte do operando. A abordagem adotada pelo VAX é, de certo modo, mais fácil para o programador, pois envolve menor número de mnemônicos. Entretanto, as instruções são menos compactas do que na abordagem adotada no IBM S/370, porque o endereço (registrador ou memória) de cada operando é especificado separadamente do código da operação. Essa discussão será retomada no próximo capítulo, ao discutirmos formatos de instruções.

Tabela 9.3 Operações comuns de conjuntos de instruções

Tipo	Nome da operação	Descrição
Operações de transferência de dados	Move	Transfere uma palavra ou bloco da fonte para o destino
	Store	Transfere uma palavra do processador para a memória
	Load	Transfere uma palavra da memória para o processador
	Exchange	Troca os conteúdos dos operandos fonte e de destino
	Clear	Transfere uma palavra contendo 0s para o destino
	Set	Transfere um palavra contendo 1s para o destino
	Push	Transfere uma palavra da fonte para o topo da pilha
	Pop	Transfere uma palavra do topo da pilha para o destino
Operações aritméticas	Add	Soma dois operandos
	Subtract	Calcula a diferença entre dois operandos
	Multiply	Calcula o produto de dois operandos
	Divide	Calcula o quociente de dois operandos
	Absolute	Substitui o operando pelo seu valor absoluto
	Negate	Muda o sinal do operando
	Increment	Soma 1 ao operando
	Decrement	Subtrai 1 do operando
Operações lógicas	AND OR NOT (Complemento) Exclusive-OR	Efetua a operação lógica especificada, bit a bit
	Test	Testa a condição especificada; atualiza códigos de condição (*flags*), de acordo com o resultado
	Compare	Efetua uma comparação lógica ou aritmética de dois ou mais operandos; atualiza códigos de condição (flags), de acordo com o resultado
	Set control variables	Classe de instruções para especificar informação de controle, para fins de proteção, tratamento de interrupção, controle de temporização etc.
	Shift	Deslocamento de operando para a esquerda (direita), introduzindo constantes no final
	Rotate	Rotação circular de operando para a esquerda (direita)
Operações de transferência de controle	Jump (branch)	Desvio incondicional; carrega o PC com o endereço especificado
	Jump conditional	Testa a condição especificada; carrega ou não o PC com o endereço especificado, conforme o resultado do teste
	Jump to subroutine	Armazena informação de controle do programa corrente em uma posição conhecida; desvia para o endereço especificado
	Return	Substitui o conteúdo do PC e de outros registradores com os valores armazenados em uma posição conhecida
	Execute	Busca o operando em uma posição especificada e executa o valor desse operando como uma instrução; não modifica o PC
	Skip	Incrementa o PC (para o endereço da próxima instrução)
	Skip conditional	Testa a condição especificada; desvia ou não com base no resultado do teste
	Halt	Pára a execução do programa
	Wait (hold)	Pára a execução do programa; testa a condição especificada repetidamente; retoma a execução quando a condição é satisfeita
	No operation	Não efetua nenhuma operação e continua a execução do programa

Tabela 9.3 Operações comuns de conjuntos de instruções (*continuação*)

Tipo	Nome da operação	Descrição
Operações de E/S	Read (input)	Transfere dados da porta ou dispositivo de E/S especificado para o destino (por exemplo, memória principal ou registrador de processador)
	Write (output)	Transfere dados da fonte especificada para uma porta ou um dispositivo de E/S
	Start I/O	Transfere instruções para o processador de E/S, para iniciar uma operação de E/S
	Test I/O	Transfere informação de estado do sistema de E/S para o destino especificado
Operações de conversão	Translate	Traduz valores armazenados em uma seção da memória, com base em uma tabela de correspondências
	Convert	Converte o conteúdo de uma palavra de uma representação para outra (por exemplo, decimal empacotado para binário)

Tabela 9.4 Ações da CPU para vários tipos de operações

Transferência de dados	Transfere dados de uma posição para outra Se envolver endereço de memória: Determina o endereço de memória Efetua conversão entre endereço de memória real e virtual Verifica a memória cache Inicia leitura/escrita na memória
Operação aritmética	Pode envolver transferência de dados, antes e/ou depois da operação Executa a operação na ULA Atualiza códigos de condição
Operação lógica	Análoga à operação aritmética
Operação de conversão	Análoga à operação aritmética ou lógica. Pode envolver lógica especial para efetuar a conversão
Transferência de controle	Atualiza o contador de programa. No caso de chamada/retorno de sub-rotina, gerencia a passagem de parâmetros e o encadeamento de chamadas
Operação de E/S	Emite comando para um módulo de E/S No caso de E/S mapeada na memória, determina o endereço mapeado na memória

As operações de transferência de dados são o tipo mais simples de operação, em termos da ação tomada pela CPU. Se o operando fonte e de destino são registradores, a CPU simplesmente transfere dados de um registrador para outro; essa é uma operação interna da CPU. Se um ou ambos os operandos estão na memória, a CPU tem de efetuar algumas ou todas as ações a seguir:

1. Calcule o endereço de memória, com base no modo de endereçamento especificado (discutido no Capítulo 10).
2. Se o endereço se refere à memória virtual, traduza esse endereço para um endereço de memória real.
3. Determine se o item endereçado está na memória cache.
4. Se não estiver, emita um comando para o módulo de memória.

Tabela 9.5 Exemplos de operações de transferência de dados do IBM S/370

Mnemônico da operação	Nome	Número de bits transferidos	Descrição
L	Load	32	Transfere da memória para o registrador
LH	Load Halfword	16	Transfere da memória para o registrador
LR	Load	32	Transfere do registrador para o registrador
LER	Load (Short)	32	Transfere do registrador de ponto flutuante para o registrador de ponto flutuante
LE	Load (Short)	32	Transfere da memória para o registrador de ponto flutuante
LDR	Load (Long)	64	Transfere do registrador de ponto flutuante para o registrador de ponto flutuante
LD	Load (Long)	64	Transfere da memória para o registrador de ponto flutuante
ST	Store	32	Transfere do registrador para a memória
STH	Store Halfword	16	Transfere do registrador para a memória
STC	Store Character	8	Transfere do registrador para a memória
STE	Store (Short)	32	Transfere do registrador de ponto flutuante para a memória
STD	Store (Long)	64	Transfere do registrador de ponto flutuante para a memória

Operações aritméticas

A maioria das máquinas fornece operações aritméticas básicas para soma, subtração, multiplicação e divisão. Essas operações são oferecidas, invariavelmente, para números inteiros com sinal (de ponto fixo). Muitas vezes, elas são também oferecidas para números na representação decimal empacotada e números de ponto flutuante.

Outras possíveis operações incluem uma variedade de instruções com um único operando. Por exemplo:

- Tomar o valor absoluto do operando.
- Negar o operando.
- Incrementar o operando de 1.
- Decrementar o operando de 1.

A execução de uma instrução aritmética pode envolver transferência de dados, para fornecer os valores dos operandos como entrada para a ULA e para armazenar na memória o valor obtido como saída da ULA. A Figura 3.5 mostra as movimentações de dados envolvidas em operações de transferência de dados e operações aritméticas. Além disso, é claro, a ULA efetua a operação desejada.

Operações lógicas

A maioria das máquinas fornece também uma variedade de operações para manipular bits individuais de uma palavra ou de qualquer unidade endereçável. Essas operações são baseadas em operações booleanas (veja o Apêndice A).

Algumas operações lógicas básicas que podem ser efetuadas sobre dados binários ou booleanos são mostradas na Tabela 9.6. A operação NOT (NÃO) inverte um bit. As operações AND (E), OR (OU) e XOR (ou-exclusivo) são as funções lógicas mais comuns com dois operandos. A operação EQUAL é um teste de igualdade binária, bastante útil.

Tabela 9.6 Operações lógicas básicas

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

Essas operações lógicas podem ser aplicadas bit a bit a unidades de dados lógicos de *n* bits. Portanto, se dois registradores contêm os dados:

```
(R1) = 10100101
(R2) = 00001111
```

então,

```
(R1) AND (R2) = 00000101
```

onde a notação (X) significa o conteúdo do endereço X. Dessa maneira, a operação AND pode ser usada como uma *máscara* para selecionar determinados bits de uma palavra, colocando zeros nos demais bits. Como outro exemplo, se dois registradores contêm os dados:

```
(R1) = 10100101
(R2) = 11111111
```

então,

```
(R1) XOR (R2) = 01011010
```

Se um dos operandos é uma palavra cujos bits são todos iguais a 1, o resultado da operação XOR é inverter todos os bits do outro operando (complemento de um).

Além das operações lógicas bit a bit, a maioria das máquinas fornece uma variedade de funções de deslocamento e rotação de bits. As operações mais básicas são mostradas na Figura 9.5. Em um *deslocamento lógico*, os bits de uma palavra são deslocados para a esquerda ou para

a direita. O bit deslocado em uma das extremidades é perdido e um zero é inserido na outra extremidade. Deslocamentos lógicos são úteis, principalmente, para isolar campos de bits dentro de uma palavra. Os 0s que são inseridos na palavra deslocam a informação não desejada, que é eliminada na outra extremidade.

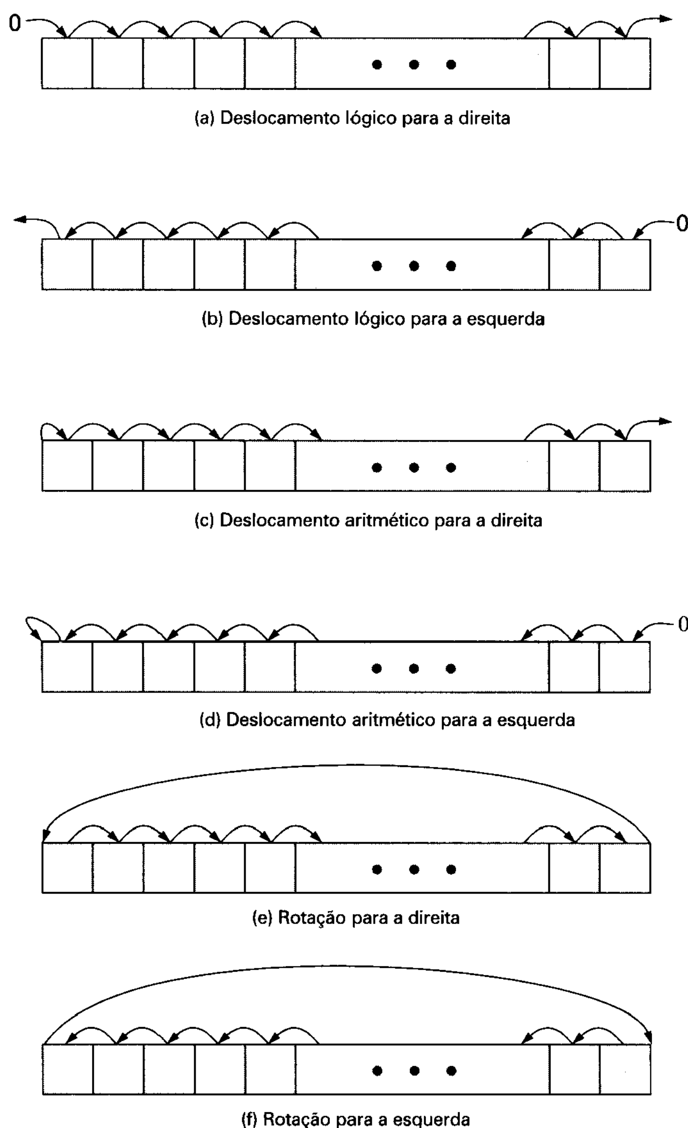


Figura 9.5 Operações de deslocamento e rotação.

Por exemplo, suponha que queremos transmitir caracteres de dados para um dispositivo de E/S, um caractere de cada vez. Se cada palavra da memória tiver tamanho de 16 bits e contiver dois caracteres, será necessário *desempacotar* os caracteres, antes que sejam enviados. Para enviar os dois caracteres de uma palavra,

1. Carregue a palavra em um registrador.
2. Execute a instrução AND com operandos dados pelo conteúdo do registrador e o valor 1111111100000000, para mascarar o caractere da direita.
3. Desloque o conteúdo do registrador para a direita, oito posições de bit. Isso desloca o caractere restante para a metade direita do registrador.
4. Efetue a E/S. O módulo de E/S lerá os 8 bits de ordem inferior do barramento de dados.

Os passos anteriores resultam no envio do caractere mais à esquerda. Para enviar o caractere mais à direita,

1. Carregue a palavra de novo no registrador.
2. Execute a instrução AND com operandos dados pelo conteúdo do registrador e o valor 0000000011111111.
3. Efetue a E/S.

A operação de *deslocamento aritmético* trata os dados como números inteiros com sinal e não altera o bit de sinal. Em um deslocamento aritmético para a direita, o bit de sinal normalmente é replicado na posição de bit à sua direita. Essas operações podem acelerar certas operações aritméticas. No caso de números representados em complemento de dois, um deslocamento para a esquerda ou para a direita corresponde, respectivamente, à multiplicação ou à divisão por 2, desde que não ocorra *overflow* ou *underflow*.

Operações de *rotação*, ou deslocamento cíclico, preservam todos os bits sobre os quais a operação é efetuada. Um possível uso da operação de rotação é para trazer sucessivamente cada bit da palavra para a posição de bit mais à esquerda, onde ele pode ser identificado por meio de um teste de sinal do dado (quando tratado como um número).

Assim como as operações aritméticas, as operações lógicas envolvem atividade da ULA e podem envolver transferência de dados.

Operações de conversão

Instruções de conversão são aquelas que mudam ou operam sobre o formato de dados. Um exemplo simples é a conversão de um número decimal para binário e um exemplo mais complexo, a instrução Translate (TR) do S/370. Essa instrução pode ser usada para converter um código de 8 bits para outro e tem três operandos:

```
TR R1, R2, L
```

O operando R2 contém o endereço do início de uma tabela de códigos de 8 bits. Os L bytes a partir do byte especificado em R1 são traduzidos, cada byte sendo substituído pelo conteúdo da entrada na tabela indexada por esse byte. Por exemplo, para traduzir de EBCDIC para ASCII, primeiramente criamos uma tabela de 256 bytes, carregada, digamos, nas posições de memória de endereço hexadecimal 1000 a 10FF. A tabela contém os caracteres do código ASCII, na seqüência em que ocorrem no código EBCDIC; ou seja, o código de um caractere ASCII é colocado na tabela na posição relativa igual ao valor binário do código EBCDIC desse mesmo caractere. Portanto, as posições 10F0 a 10F9 conterão os valores 30 a 39, porque F0 é o código EBCDIC para o dígito 0 e 30 é o código ASCII para o dígito 0, e assim por diante, até o dígito 9. Suponha agora que os dígitos 1984, codificados em EBCDIC, estejam armazenados a partir do endereço 2100 e que queremos traduzir esses dígitos para ASCII. Considere o seguinte:

- As posições de endereço 2100-2103 contêm F1 F9 F8 F4.
- R1 contém 2100.
- R2 contém 1000.

Então, se executarmos:

```
TR R1, R2, 4
```

as posições 2100-2103 conterão os valores 31 39 38 34.

Operações de entrada/saída

As instruções de entrada/saída (E/S) foram discutidas com algum detalhe no Capítulo 6. Como vimos, existe uma variedade de abordagens, incluindo E/S programada, E/S mapeada na memória, DMA e uso de processadores de E/S. Muitas implementações fornecem apenas algumas instruções de E/S, com ações específicas determinadas por meio de parâmetros, códigos ou palavras de comando.

Operações de controle de sistema

Instruções de controle de sistema são aquelas que apenas podem ser executadas quando o processador está no estado privilegiado ou está executando um programa carregado em uma área especial da memória, que é privilegiada. Tipicamente, elas são reservadas para uso pelo sistema operacional.

Alguns exemplos de operações de controle de sistema são dados a seguir. Uma instrução de controle de sistema pode servir para ler ou modificar o conteúdo de um registrador de controle; os registradores de controle serão discutidos no Capítulo 11. Outro exemplo seria uma instrução para ler ou modificar uma chave de proteção de memória, tal como é usado no sistema de memória do IBM S/370. Outro exemplo, ainda, seria o acesso a blocos de controle de processo, em um sistema de multiprogramação.

Operações de transferência de controle

Para todos os tipos de operação discutidos até agora, a próxima instrução a ser executada é aquela que segue imediatamente, na memória, a instrução corrente. No entanto, uma fração significativa das instruções de qualquer programa tem como função alterar a seqüência de execução de instruções. Nessas instruções, a CPU atualiza o contador de programa com o endereço de alguma outra instrução armazenada na memória.

Operações de transferência de controle são requeridas por diversas razões. Entre as mais importantes estão as seguintes:

1. No uso prático de computadores, é essencial poder executar um conjunto de instruções mais de uma vez e talvez milhares de vezes. Milhares ou talvez milhões de instruções podem ser requeridas para implementar uma aplicação. Isso seria impensável se cada instrução tivesse de ser escrita separadamente. O processamento de uma tabela ou lista de dados, por exemplo, pode ser feito ao executar repetidamente uma seqüência de instruções para processar cada item de dado.
2. Quase todos os programas envolvem a tomada de algumas decisões, isto é, o computador deve executar uma determinada seqüência de operações, se uma determinada condição é satisfeita, e uma outra seqüência de operações, se essa condição não se verifica.

Por exemplo, considere uma seqüência de instruções para computar a raiz quadrada de um número. No início dessa seqüência, o sinal do número pode ser testado e, caso o número seja negativo, a computação não é efetuada, sendo reportada uma condição de erro.

3. Implementar corretamente um programa de computador de grande porte, ou mesmo de tamanho médio, é uma tarefa excessivamente difícil. Por isso, é útil dispor de mecanismos para dividir o programa em partes menores, que possam ser programadas separadamente.

A seguir, discutimos as operações de transferência de controle encontradas mais comumente em um conjunto de instruções: as operações de desvio, de salto e de chamada de procedimento.

Instruções de desvio

Uma instrução de desvio tem como um de seus operandos o endereço da próxima instrução a ser executada. Com freqüência, essa instrução é um *desvio condicional*, isto é, o desvio será feito (o contador de programa é atualizado com o endereço especificado no operando) apenas se uma dada condição for satisfeita. Caso contrário, será executada a próxima instrução da seqüência de instruções (o contador de programa é incrementado).

Existem duas formas comuns de gerar a condição a ser testada em uma instrução de desvio condicional. Primeiramente, a maioria das máquinas contém um código de condição, de 1 bit ou de múltiplos bits, que é atualizado de acordo com o resultado de determinadas operações. Esse código pode ser visto como um pequeno registrador, visível para o usuário. Por exemplo, uma operação aritmética (tal como a soma ou a subtração) pode atualizar um código de condição de 2 bits, com um dos quatro seguintes valores: 0, positivo, negativo, *overflow*. Uma máquina com esse código de condição poderia ter quatro tipos de desvio condicional:

```
BRP X   Desviará para a instrução de endereço X se o resultado for positivo
BRN X   Desviará para a instrução de endereço X se o resultado for negativo
BRZ X   Desviará para a instrução de endereço X se o resultado for zero
BRO X   Desviará para a instrução de endereço X se ocorrer overflow
```

Em todos esses casos, a condição se refere ao resultado da última operação que atualizou o código de condição.

Outra abordagem possível, para um formato de instrução com três endereços, é especificar uma operação de comparação e um desvio na mesma instrução. Por exemplo,

```
BRE R1, R2, X   Desviará para a instrução de endereço X se o conteúdo
                de R1 for igual ao conteúdo de R2
```

A Figura 9.6 apresenta exemplos dessas operações. Note que um desvio pode ser tanto *para a frente* (para uma instrução de endereço mais alto) como *para trás* (para uma instrução de endereço mais baixo). O exemplo mostra como instruções de desvio condicional e de desvio incondicional podem ser usadas para implementar um laço de repetição, isto é, uma seqüência de instruções que são executadas repetidamente. As instruções de endereço 202 a 210 são executadas repetidamente, até que o resultado da subtração X – *Y seja igual a 0.*

Instruções de salto

Outra forma comum de instrução de transferência de controle é a instrução de salto. Instruções desse tipo incluem um endereço de desvio implícito. Tipicamente, um salto indica que a execução de uma instrução da seqüência de instruções deve ser omitida; portanto, o ende-

reço da próxima instrução a ser executada é obtido somando o endereço da instrução corrente com o tamanho de uma instrução.

Como instruções de salto não requerem um campo de endereço de instrução, a área correspondente a esse campo pode ser usada para informações de outra natureza. Um exemplo típico é uma instrução para incrementar o valor contido em um registrador e saltar caso o resultado dessa operação seja igual a zero (ISZ — *increment-and-skip-if-zero*). Considere o seguinte fragmento de programa:

```
301
•
•
•
309 ISZ R1
310 BR 301
311
```

Nesse fragmento, duas instruções de transferência de controle são usadas para implementar um laço de repetição. O registrador R1 é inicializado com o número de iterações a serem efetuadas, com sinal negativo. No final do laço, o conteúdo de R1 é incrementado. Se o valor obtido for diferente de 0, a execução do programa desviará de volta para o início do laço. Caso contrário, a execução dessa instrução de desvio será omitida e a execução do programa continuará com a instrução seguinte ao final do laço.

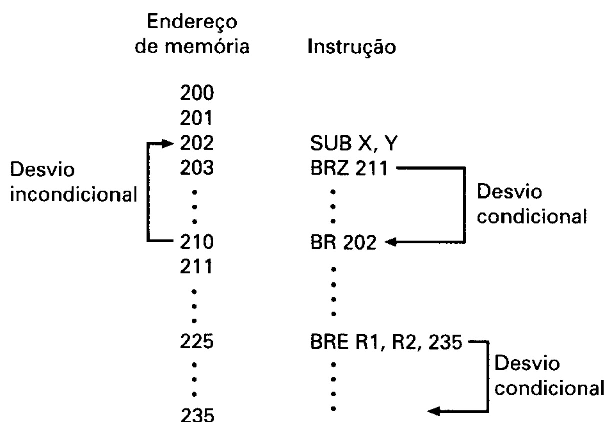


Figura 9.6 Instruções de desvio.

Instruções de chamada de procedimento

O conceito de *procedimento* foi talvez uma das mais importantes inovações no desenvolvimento de linguagens de programação. Um procedimento é um subprograma autocontido, que é incorporado em um programa maior. Um procedimento pode ser invocado, ou *chamado*, em qualquer ponto do programa. Uma chamada a um procedimento instrui o processador a executar todo o procedimento e, então, retornar ao ponto em que ocorreu a chamada.

As duas principais razões para usar procedimentos são economia e modularidade. O uso de procedimentos permite que um mesmo trecho de código seja usado muitas vezes. Isso é importante para a economia de esforço de programação e para o uso mais eficiente do es-

paço de armazenamento do sistema (uma vez que programas devem ser armazenados). Além disso, os procedimentos possibilitam a divisão de grandes tarefas de programação em unidades menores. Esse uso de *modularidade* facilita muito a tarefa de programação.

O mecanismo de controle de procedimentos envolve duas instruções básicas: uma instrução de chamada, que desvia a execução da instrução corrente para o início do procedimento, e uma instrução de retorno, que provoca o retorno da execução do procedimento para o endereço em que ocorreu a chamada. Ambas constituem formas de instrução de desvio.

A Figura 9.7a mostra o uso de procedimentos para construir um programa. Nesse exemplo, o programa principal é carregado a partir do endereço 4000. O programa inclui uma chamada ao procedimento *PROC1*, que inicia no endereço 4500. Quando essa instrução de chamada é executada, a CPU suspende a execução do programa principal e inicia a execução de *PROC1*, buscando a próxima instrução no endereço 4500. Dentro de *PROC1*, existem duas chamadas ao procedimento *PROC2*, armazenadas a partir do endereço 4800. Em cada chamada, a execução de *PROC1* é suspensa e *PROC2* é executado. O comando RETURN faz com que a CPU retorne para o programa que efetuou a chamada, continuando com a execução da instrução imediatamente seguinte à instrução CALL correspondente. Esse comportamento é mostrado na Figura 9.7b.

Diversos pontos devem ser notados:

1. Um procedimento pode ser chamado de mais de um ponto do programa.
2. Uma chamada a um procedimento pode ocorrer dentro de um procedimento. Isso possibilita o *aninhamento* de procedimentos, até uma profundidade arbitrária.
3. A cada chamada de procedimento corresponde um retorno, para o programa que efetuou a chamada.

Como um procedimento pode ser chamado de diversos pontos de um programa, o endereço de retorno deve ser salvo pela CPU de alguma maneira, para que o retorno possa ser feito adequadamente. É comum armazenar o endereço de retorno nos seguintes locais:

- Em um registrador
- No início da área de memória do procedimento
- No topo da pilha

Considere uma instrução CALL X, em linguagem de máquina, que representa uma *chamada ao procedimento de endereço X*. Se o endereço de retorno for armazenado em um registrador, a instrução CALL X causará as seguintes ações:

$$RN \leftarrow PC + \Delta$$
$$PC \leftarrow X$$

onde RN é o registrador usado para armazenar o endereço de retorno da chamada de procedimento, PC é o contador de programa e Δ é o tamanho de uma instrução. O procedimento chamado pode, então, salvar o conteúdo de RN, para que ele seja usado posteriormente, no retorno do procedimento.

Uma segunda possibilidade é armazenar o endereço de retorno no início da área de memória do procedimento. Nesse caso, a instrução CALL X causará as seguintes ações:

$$X \leftarrow PC + \Delta$$
$$PC \leftarrow X + 1$$

Isso é bastante conveniente, pois o endereço de retorno é armazenado com segurança.

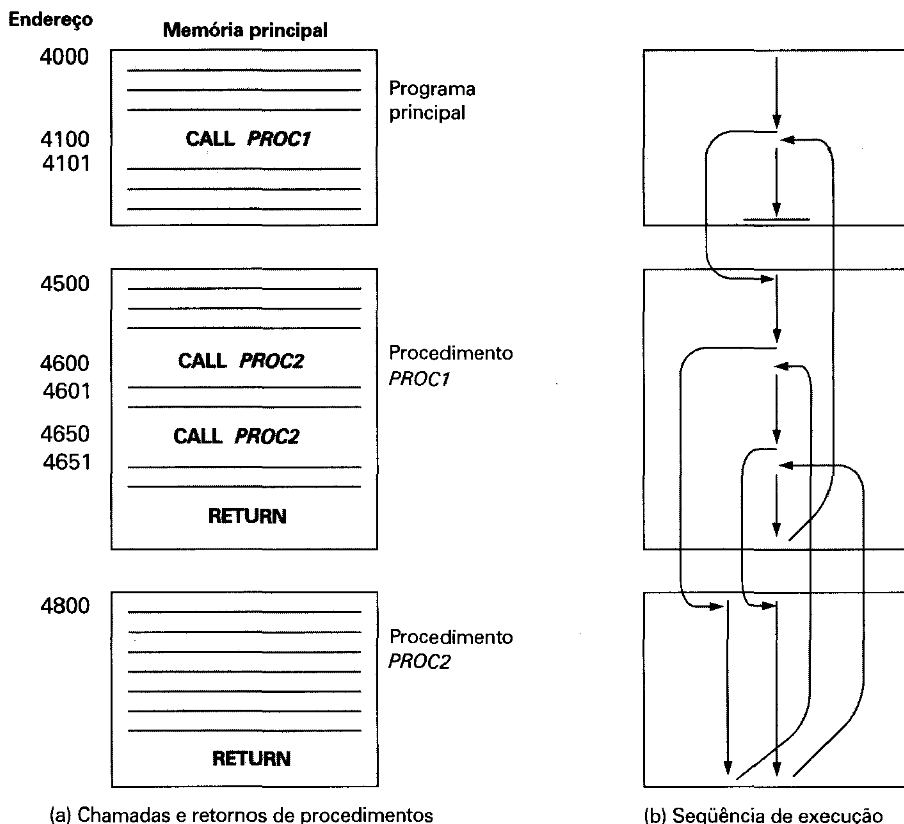


Figura 9.7 Procedimentos aninhados.

As duas abordagens anteriores funcionam corretamente e têm sido usadas. A única limitação dessas abordagens é que impedem o uso de procedimentos *reentrantes*. Um procedimento será reentrante se for possível ter várias chamadas para esse procedimento ao mesmo tempo. Um exemplo do uso dessa característica é um procedimento recursivo (que chama a si próprio).

Uma abordagem mais geral e poderosa consiste em usar uma pilha para armazenar o endereço de retorno de uma chamada de procedimento (veja o Apêndice 9A para uma discussão sobre pilhas). Quando a chamada é executada, a CPU coloca o endereço de retorno na pilha. Quando é executado o retorno, o endereço armazenado na pilha é usado. A Figura 9.8 mostra o uso da pilha.

Em uma chamada de procedimento, além de fornecer o endereço de retorno, é muitas vezes necessário passar também parâmetros para o procedimento. Esses parâmetros podem ser passados em registradores. Uma outra possibilidade consiste em armazenar os parâmetros na memória, logo depois da instrução de chamada. Nesse caso, o retorno do procedimento deve ser feito para o endereço seguinte à área de parâmetros. Essas duas abordagens têm algumas desvantagens. Se os parâmetros forem passados em registradores, o programa que efetuou a chamada e o procedimento chamado deverão assegurar que os registradores sejam usados adequadamente. O armazenamento de parâmetros na memória faz com que seja difícil passar um número variável de parâmetros. Ambas as abordagens impedem o uso de procedimentos reentrantes.

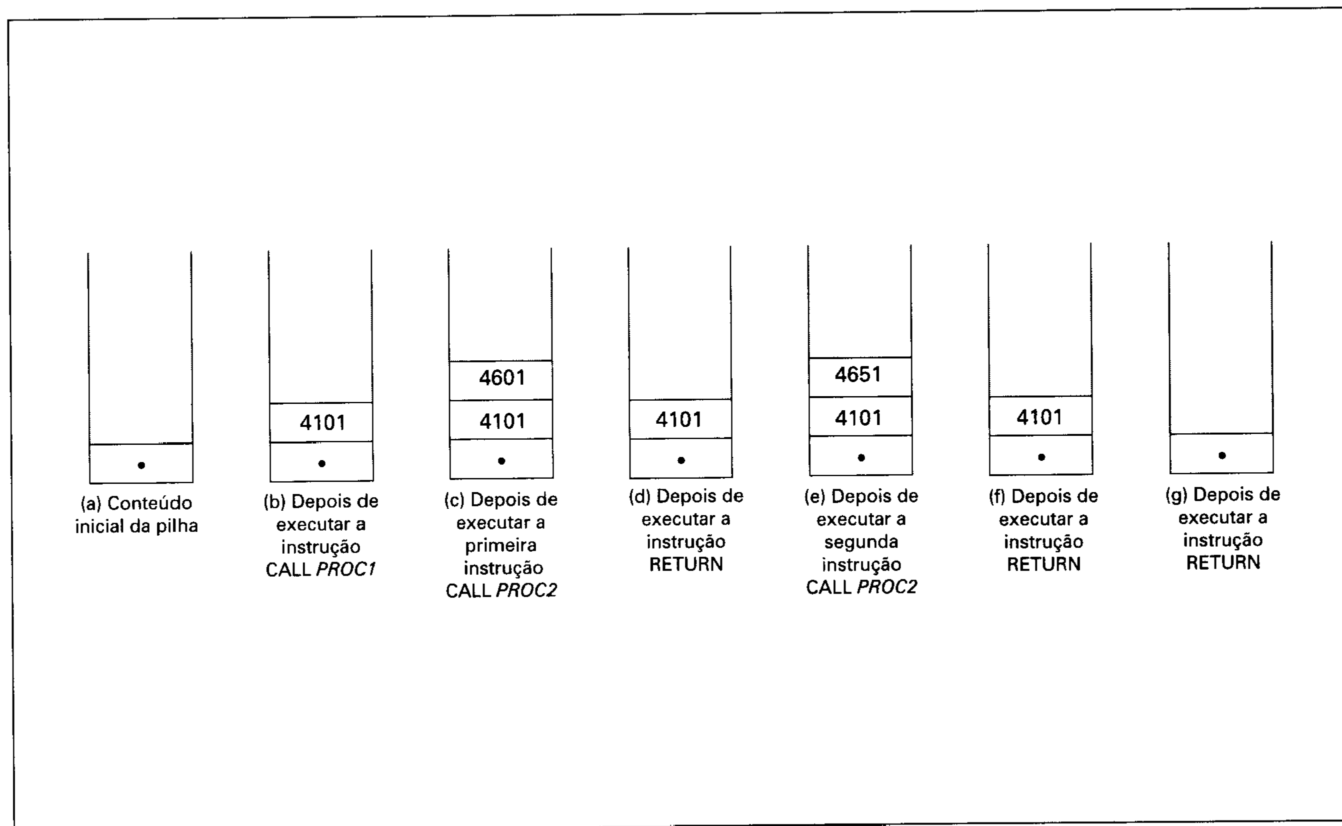


Figura 9.8 Uso da pilha para implementar os procedimentos aninhados da Figura 9.7.

Uma abordagem mais flexível é usar a pilha para a passagem de parâmetros. Quando o processador executa uma chamada, não apenas é empilhado o endereço de retorno mas também os parâmetros a serem passados para o procedimento. O procedimento chamado pode endereçar os parâmetros diretamente na pilha. No retorno do procedimento, os parâmetros de saída podem também ser armazenados na pilha. O conjunto de dados armazenados na pilha em uma chamada de procedimento, incluindo parâmetros e endereço de retorno, é conhecido como *registro de ativação.*

Um exemplo é mostrado na Figura 9.9. O exemplo refere-se a um procedimento P, onde são declaradas as variáveis locais *x*1 e *x*2, e a um procedimento Q, que pode ser chamado por P, e no qual são declaradas as variáveis locais *y*1 e *y*2. O endereço de retorno de cada chamada de procedimento é o primeiro item armazenado no registro de ativação correspondente. Em seguida, é armazenado um apontador para o início do registro de ativação anterior. Isso é necessário se o número ou o tamanho dos parâmetros a serem empilhados é variável.

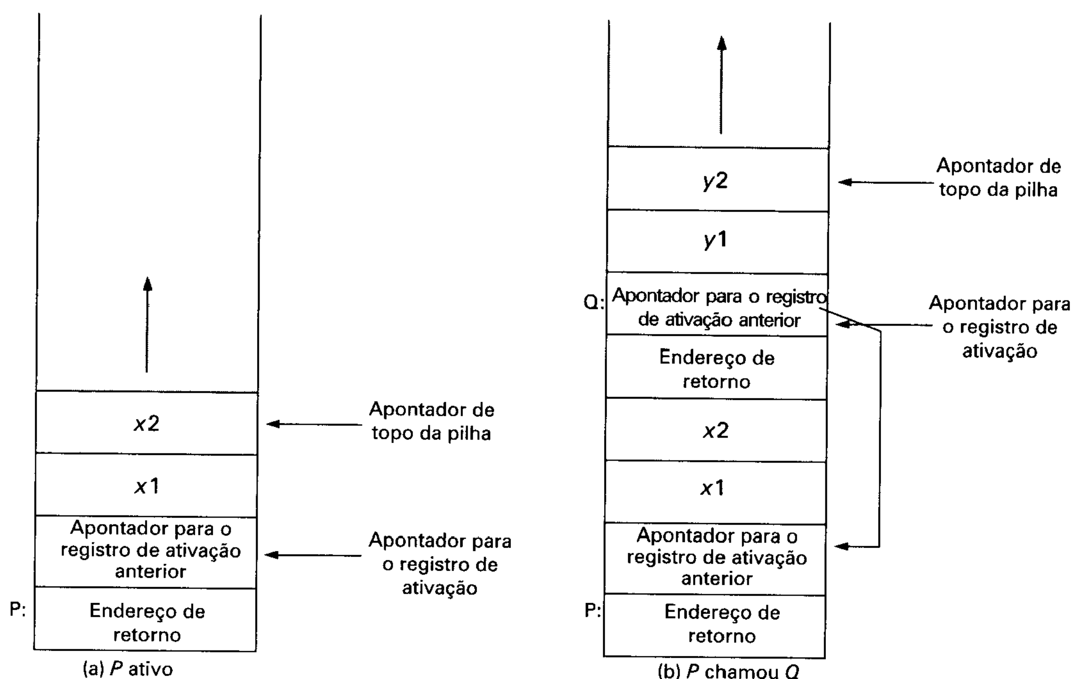


Figura 9.9 Crescimento da pilha de chamada de procedimentos (Dewar, 1990).

9.5 TIPOS DE OPERAÇÕES DO PENTIUM II E DO PowerPC

Tipos de operações do Pentium II

O Pentium II oferece um complexo conjunto de tipos de operações, incluindo diversas instruções especializadas. No projeto do conjunto de instruções do Pentium II, a intenção consistia em fornecer ferramentas para facilitar a implementação de compiladores capazes de gerar código de máquina otimizado, a partir de programas em linguagem de alto nível. A Tabela

9.7 apresenta os tipos de instrução disponíveis, dando exemplos de cada um. A maioria delas consiste em instruções convencionais, encontradas em grande parte dos conjuntos de instruções de máquina, mas existem diversos tipos de instrução projetados especialmente para a arquitetura 80x86/Pentium, que são de interesse específico.

Tabela 9.7 Tipos de operações do Pentium II (incluindo exemplos de operações típicas)

Instrução	Descrição
	Instruções de transferência de dados
MOV	Move operando, entre registradores ou entre registrador e memória.
PUSH	Coloca o operando no topo da pilha.
PUSHA	Coloca todos os registradores na pilha.
MOVSX	Move byte, palavra ou palavra dupla, com extensão de sinal. Move um byte para uma palavra ou uma palavra para uma palavra dupla, com extensão de sinal, usando representação em complemento de dois.
LEA	Carrega endereço efetivo. Carrega o endereço do operando fonte e não o seu valor no operando destino.
XLAT	Tradução baseada em tabela. Substitui o byte em AL pelo byte correspondente de uma tabela de tradução, codificada pelo usuário. Quando a instrução XLAT é executada, AL deve conter um índice (número inteiro sem sinal) para a tabela. O conteúdo de AL é alterado para o conteúdo da entrada na tabela correspondente a esse índice.
IN, OUT	Efetua uma operação de entrada ou saída, no espaço de endereçamento de E/S.
	Instruções aritméticas
ADD	Soma dos operandos.
SUB	Subtração dos operandos.
MUL	Multiplicação de números inteiros sem sinal, com operandos com tamanho de byte, palavra ou palavra dupla, e resultado com tamanho de palavra, palavra dupla ou palavra quádrupla.
IDIV	Divisão de números inteiros com sinal.
	Instruções lógicas
AND	Executa a operação E lógico dos operandos.
BTS	Testa e altera valor de um bit. O operando é um campo de bits. A instrução copia o valor corrente de um bit para o *flag* CF do registrador de *flags* EF e altera o valor do bit original para 1.
BSF	Pesquisa por um bit com valor 1, em uma palavra ou palavra dupla, e armazena o número do primeiro bit com valor 1 encontrado em um registrador.
SHL/SHR	Deslocamento lógico para a esquerda e para a direita.
SAL/SAR	Deslocamento aritmético para a esquerda e para a direita.
ROL/ROR	Rotação para a esquerda e para a direita.
SETcc	Altera o valor de um byte para 0 ou 1, dependendo de qualquer uma das 16 condições indicadas pelos *flags* de estado.

(*continua*)

Tabela 9.7 Tipos de operações do Pentium II (incluindo exemplos de operações típicas) (*continuação*)

Instrução	Descrição
	Instruções de transferência de controle
JMP	Desvio incondicional.
CALL	Transfere o controle para um procedimento. Antes de transferir o controle, o endereço da instrução que segue a instrução CALL é armazenado na pilha.
JE/JZ	Desvia se igual/Desvia se zero.
LOOPE/LOOPZ	Repete se igual/Repete se zero. É uma instrução de desvio condicional, que usa o valor armazenado no registrador ECX. A instrução primeiro decrementa o conteúdo de ECX, antes de usá-lo para o teste especificado como condição de desvio.
INT/INTO	Interrupção/Interrupção se *overflow*. Transfere o controle para uma rotina de tratamento de interrupção.
	Operações sobre seqüências de caracteres
MOVS	Move byte, palavra ou palavra dupla de uma seqüência. A instrução opera sobre um elemento de uma seqüência indexada pelos registradores ESI e EDI. Depois de cada operação, os registradores são incrementados ou decrementados automaticamente, para apontar para o próximo elemento da seqüência.
LODS	Carrega byte, palavra ou palavra dupla de uma seqüência.
	Suporte para linguagem de alto nível
ENTER	Cria um registro de ativação na pilha, que pode ser usada para implementar regras de uma linguagem de alto nível estruturada em blocos.
LEAVE	Reverte a ação de uma instrução ENTER anterior.
BOUND	Verifica limites de vetor. Verifica se o valor contido no operando 1 está compreendido entre um limite inferior e um limite superior. Esses limites devem estar armazenados em duas posições de memória adjacentes, referenciadas pelo operando 2. Caso o valor esteja fora dos limites, ocorre uma interrupção. Essa instrução é usada para verificar se um índice de um vetor está dentro dos limites da área alocada para o vetor.
	Operações sobre bits de condição
STC	Ativa o bit CF ('vai-um') do registrador de *flags*.
LAHF	Copia os bits SF, ZF, AF, PF e CF do registrador AH.
	Operações sobre registradores de segmento
LDS	Carrega um apontador de segmento no registrador de segmento DS.
	Operações de controle de sistema
HLT	Pára a execução do programa (suspende o processador).
LOCK	Bloqueia uma área da memória compartilhada, para que o Pentium II tenha uso exclusivo dessa área durante a instrução seguinte ao LOCK.
ESC	Escape para uma extensão do processador. Código de escape que indica que as instruções seguintes devem ser executadas por um co-processador numérico, que fornece operações sobre números inteiros de alta precisão e números ponto flutuante.
WAIT	Espera até que o sinal BUSY# seja negado. Suspende a execução do programa até que o processador detecte que o pino BUSY está inativo, indicando que o co-processador numérico terminou a execução de uma operação.

Tabela 9.7 Tipos de operações do Pentium II (incluindo exemplos de operações típicas) (*continuação*)

Instrução	Descrição
	Operações de proteção
SGDT	Armazena a tabela de descritor global.
LSL	Carrega limite de segmento. Carrega um registrador especificado pelo usuário com um limite de segmento.
VERR/VERW	Verifica segmento para leitura/escrita.
	Operações de gerenciamento da memória cache
INVD	Invalida a memória cache interna.
WBINVD	Invalida a memória cache interna, depois de gravar na memória as linhas da cache que foram alteradas.
INVLPG	Invalida uma entrada da cache de tradução de endereços (TLB).

Instruções de chamada/retorno de procedimento

O Pentium II oferece quatro instruções para suporte a chamada/retorno de procedimento: CALL, ENTER, LEAVE, RETURN. Da Figura 9.9, lembre-se de que um modo comum de implementar o mecanismo de chamada/retorno de procedimentos é por meio de registros de ativação criados na pilha. Quando um novo procedimento é chamado, as seguintes ações devem ser executadas na entrada do novo procedimento:

- Empilhar o endereço de retorno.
- Empilhar o apontador do registro de ativação corrente.
- Copiar o apontador de topo da pilha como o novo valor do apontador para o registro de ativação.
- Ajustar o apontador de topo da pilha para alocar um registro de ativação.

A instrução CALL empilha o valor corrente do contador de programa e provoca o desvio para o endereço de início do procedimento, colocando esse endereço no contador de programa. Nas máquinas 8088 e 8086, um procedimento típico começa com a seguinte seqüência de instruções:

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, espaço_ para_variáveis_locais
```

onde EBP é o apontador para o registro de ativação e ESP é o apontador de topo da pilha. Nas máquinas 80286 e nas posteriores, a instrução ENTER efetua todas as operações acima, em uma única instrução.

A instrução ENTER foi adicionada ao conjunto de instruções para oferecer suporte direto para o compilador. Ela inclui também uma característica especial para suporte a procedimentos aninhados, que ocorrem em linguagens, tais como Pascal, COBOL e Ada (não encontrados em C ou FORTRAN). Atualmente, sabe-se que existem formas melhores de manipular chamadas de procedimentos aninhados. Além disso, embora a instrução ENTER economize alguns bytes de memória, se comparada à seqüência de instruções PUSH, MOV, SUB (4 bytes, em vez de 6 bytes), sua execução consome mais tempo (dez ciclos de relógio, em vez

de seis ciclos de relógio). Portanto, embora possa ter parecido aos projetistas do conjunto de instruções que seria uma boa idéia adicionar essa característica, essa decisão complicou a implementação do processador e resultou em pouco ou nenhum benefício. Veremos que, na abordagem RISC, ao contrário, o projeto do processador deve evitar instruções complexas, tais como ENTER, possibilitando uma implementação mais eficiente, com uma seqüência de instruções mais simples.

Gerenciamento de memória

Outro conjunto de instruções especializadas tem como finalidade lidar com a segmentação de memória. Essas instruções são privilegiadas e apenas podem ser executadas pelo sistema operacional. Elas possibilitam carregar e ler tabelas de segmentos locais e globais (chamadas de tabelas de descritores), assim como verificar ou alterar o privilégio de um segmento.

As instruções especiais para lidar com a memória cache interna do processador foram discutidas no Capítulo 4.

Códigos de condição

Mencionamos anteriormente que os códigos de condição são bits de registradores especiais, que podem ser atualizados em determinadas operações e testados em instruções de desvio condicional. Esses códigos de condição são atualizados por operações aritméticas e de comparação. Na maioria das linguagens, a operação de comparação subtrai seus dois operandos, tal como uma operação de subtração. A diferença é que a operação de comparação atualiza apenas os códigos de condição, enquanto a operação de subtração também armazena o resultado da subtração no operando de destino.

A Tabela 9.8 apresenta os códigos de condição usados no Pentium II. Cada condição, ou combinação dessas condições, pode ser testada em uma instrução de desvio condicional. A Tabela 9.9 mostra as combinações de condições para as quais são definidos códigos de operação de desvio condicional.

Tabela 9.8 Códigos de condição do Pentium II

Bit de estado	Nome	Descrição
C	'Vai-um'	Indica a ocorrência de um 'vai-um' ou de um 'vem-um' na posição de bit mais à esquerda em uma operação aritmética. É também atualizado por algumas operações de deslocamento e de rotação.
P	Paridade	Paridade do resultado de uma operação lógica ou aritmética. 1 indica paridade par; 0 indica paridade ímpar.
A	'Vai-um' auxiliar	Representa um 'vai-um' ou um 'vem-um' entre metades de um byte, em uma operação lógica ou aritmética de 8 bits, que usa o registrador AL.
Z	Zero	Indica que o resultado de uma operação lógica ou aritmética é 0.
S	Sinal	Indica o sinal do resultado de uma operação lógica ou aritmética.
O	*Overflow*	Indica a ocorrência de *overflow* em uma operação aritmética de adição ou subtração.

Diversas observações interessantes podem ser feitas sobre essa lista de condições. Primeiramente, note que um teste para determinar se um número é maior que outro depende do fato de se os operandos são tratados como números com ou sem sinal. Por exemplo, o número de 8 bits 11111111 será maior que 00000000, se ambos forem interpretados como números inteiros sem sinal (255 > 0), mas será menor, se ambos forem considerados números de 8 bits, representados em complementos de dois (–1 < 0). Por isso, muitas linguagens de montagem introduzem dois conjuntos de termos para distinguir esses dois casos: quando comparamos dois números como números inteiros com sinal, usamos os termos *menor que* e *maior que*; quando comparamos esses números como números inteiros sem sinal, usamos os termos *abaixo* e *acima*.

Tabela 9.9 Códigos de condição do Pentium II para desvio condicional e instruções SETcc

Símbolo	**Condição testada**	**Comentário**
A, NBE	C=0 AND Z=0	Acima; não abaixo ou igual (maior que, sem sinal)
AE, NB, NC	C=0	Acima ou igual; não abaixo (maior que ou igual, sem sinal); sem 'vai-um'
B, NAE, C	C=1	Abaixo; não acima ou igual (menor que, sem sinal); bit 'vai-um' igual a 1
BE, NA	C=1 OR Z=1	Abaixo ou igual; não acima (menor que ou igual, sem sinal)
E, Z	Z=1	Igual; zero (com ou sem sinal)
G, NLE	[(S=1 AND O=1) OR (S=0 AND O=0)] AND [Z=0]	Maior que; não menor que ou igual (com sinal)
GE, NL	(S=1 AND O=1) OR (S=0 AND O=0)	Maior que ou igual; não menor que (com sinal)
L, NGE	(S=1 AND O=0) OR (S=0 AND O=1)	Menor que; não maior que ou igual (com sinal)
LE, NG	(S=1 AND O=0) OR (S=0 AND O=1) OR (Z=1)	Menor que ou igual; não maior que (com sinal)
NE, NZ	Z=0	Não igual; não zero (com ou sem sinal)
NO	O=0	Não ocorreu *overflow*
NS	S=0	Sinal positivo (não negativo)
NP, PO	P=0	Sem paridade; paridade ímpar
O	O=1	*Overflow*
P	P=1	Paridade; paridade par
S	S=1	Sinal (negativo)

Uma segunda observação diz respeito à complexidade da comparação de números inteiros com sinal. Um número sem sinal será maior ou igual a zero se (1) o bit de sinal for zero e não houver *overflow* (S = 0 AND O = 0) ou (2) o bit de sinal for um 1 e houver *overflow*. Um estudo da Figura 8.5 deverá convencê-lo de que as condições testadas para as várias operações sobre números com sinal são adequadas (veja o Exercício 9.12).

Instruções do Pentium II MMX

Em 1996, a Intel introduziu a tecnologia MMX em sua linha de produtos Pentium. MMX é um conjunto de instruções altamente otimizadas para tarefas multimídia. Foram introduzidas 57 novas instruções, que tratam dados de modo SIMD (uma instrução, múltiplos dados), o que torna possível efetuar uma mesma operação, tal como adição ou multiplicação, sobre múltiplos dados de uma vez. Tipicamente, a execução de cada instrução consome um único ciclo de relógio. Para aplicações adequadas, os algoritmos que usam essas operações paralelas rápidas podem obter um desempenho duas a oito vezes maior do que a de algoritmos que não usam instruções MMX (Atkins, 1996).

O foco do MMX é a programação multimídia. Dados de vídeo e de áudio são tipicamente compostos de grandes vetores de dados de tamanho pequeno, de 8 ou 16 bits, enquanto instruções convencionais são projetadas para operar sobre dados de 32 ou 64 bits. Alguns exemplos são: em imagens e vídeos, uma única cena consiste em um vetor de pixels[1], existindo 8 bits para cada pixel ou 8 bits para cada componente de cor de um pixel (vermelho, verde, azul). Amostras típicas de áudio são codificadas usando 16 bits. Para alguns algoritmos que manipulam imagens 3D, é comum usar 32 bits para tipos de dados básicos. Para fornecer operações paralelas sobre dados com esses tamanhos, três novos tipos de dados são definidos no MMX. Cada tipo de dado tem tamanho de 64 bits e é constituído de múltiplos campos de dado menores, cada qual contendo um número inteiro de ponto fixo. Os novos tipos são:

- **Pacote de bytes:** oito bytes empacotados em quantidades de 64 bits
- **Pacote de palavras:** quatro palavras de 16 bits empacotadas em 64 bits
- **Pacote de palavras duplas:** duas palavras duplas de 32 bits empacotadas em 64 bits

A Tabela 9.10 mostra o conjunto de instruções MMX. A maioria das instruções envolve a operação paralela sobre bytes, palavras ou palavras duplas. Por exemplo, a instrução PSLLW efetua um deslocamento lógico para a esquerda, separadamente, em cada uma das quatro palavras do pacote de palavras do operando; a instrução PADDB tem pacotes de bytes como operandos de entrada e efetua adições paralelas em cada posição de byte, independentemente, produzindo como saída um pacote de bytes.

1. Um pixel (do inglês, *picture element*), ou elemento de figura, é o menor elemento de uma imagem digital, ao qual pode ser atribuído um nível de cinza. De maneira equivalente, ele é um ponto individual em uma matriz de representação de pontos de uma figura.

Tabela 9.10 Conjunto de instruções MMX

Categoria	Instrução	Descrição
Instruções aritméticas	PADD [B, W, D]	Adiciona, em paralelo, um pacote de oito bytes ou de quatro palavras de 16 bits ou de duas palavras duplas de 32 bits, e usa truncamento.
	PADDS [B, W]	Adiciona e usa saturação.
	PADDUS [B, W]	Adiciona números sem sinal e usa saturação.
	PSUB [B, W, D]	Subtrai e usa truncamento.
	PSUBS [B, W]	Subtrai com saturação.
	PSUBUS [B, W]	Subtrai números sem sinal e usa saturação.
	PMULHW	Multiplica, em paralelo, quatro palavras de 16 bits, com sinal, e dá como resultado os 16 bits mais significativos do resultado de 32 bits.
	PMULLW	Multiplica, em paralelo, quatro palavras de 16 bits com sinal, e dá como resultado os 16 bits menos significativos do resultado de 32 bits.
	PMADDWD	Multiplica, em paralelo, quatro palavras de 16 bits com sinal; soma os pares adjacentes de resultados de 32 bits.
Instruções de comparação	PCMPEQ [B, W, D]	Teste de igualdade, em paralelo; o resultado é uma máscara de 1s, se verdadeiro, ou de 0s, se falso.
	PCMPGT [B, W, D]	Teste de maior que, em paralelo; o resultado é uma máscara de 1s, se verdadeiro, ou de 0s, se falso.
Instruções de conversão	PACKUSWB	Empacota palavras em bytes e usa saturação sem sinal.
	PACKSS [WB, DW]	Empacota palavras em bytes ou palavras duplas em palavras e usa saturação com sinal.
	PUNPCKH [BW, WD, DQ]	Desempacota em paralelo (intercalação) bytes, palavras ou palavras duplas de ordem superior de um registrador MMX.
	PUNPCKL [BW, WD, DQ]	Desempacota, em paralelo (intercalação), bytes, palavras ou palavras duplas de ordem inferior de um registrador MMX.

(*continua*)

Tabela 9.10 Conjunto de instruções MMX (*continuação*)

Categoria	Instrução	Descrição
Instruções lógicas	PAND	Operação lógica AND, bit a bit, de 64 bits.
	PNDN	Operação lógica AND NOT, bit a bit, de 64 bits.
	POR	Operação lógica OR, bit a bit, de 64 bits.
	PXOR	Operação lógica XOR, bit a bit, de 64 bits.
Instruções de deslocamento	PSLL [W, D, Q]	Deslocamento lógico para a esquerda, em paralelo, de um pacote de palavras, palavras duplas ou palavras quádruplas, de um número de posições especificado por um registrador MMX ou por um operando imediato.
	PSRL [W, D, Q]	Deslocamento lógico para a direita, em paralelo, de um pacote de palavras, palavras duplas ou palavras quádruplas.
	PSRA [W, D]	Deslocamento aritmético para a direita, em paralelo, de um pacote de palavras, palavras duplas ou de uma palavra quádrupla.
Transferência de dados	MOV [D, Q]	Move palavra dupla ou palavra quádrupla de/para registrador MMX.
Gerência de estado	EMMS	Esvazia estado MMX (esvaziar bits de condição dos registradores FP).
Nota: Se uma instrução oferecer suporte para múltiplos tipos de dados [byte (B), palavra (W), palavra dupla (D), palavra quádrupla (Q)], os tipos de dados serão indicados entre colchetes.		

Uma característica incomum do novo conjunto de instruções é a introdução de aritmética de saturação. Na aritmética usual de números sem sinal, quando ocorre *overflow* (por exemplo, 'vai-um' no bit mais significativo), o bit extra é truncado. O efeito desse truncamento pode, por exemplo, produzir um resultado de uma adição que é menor que os dois operandos de entrada. Considere a adição de palavras com valor hexadecimal F000h e 3000h. A soma seria expressa como:

```
 F000h  = 1111 0000 0000 0000
+3000h  = 1111 0000 0000 0000
         10010 0000 0000 0000 = 2000h
```

Se esses dois números representam intensidade de imagens, o resultado dessa adição faz com que a combinação de duas sombras escuras produza uma sombra mais clara. Isso não é normalmente o que se pretende. Na aritmética de saturação, se uma adição resulta em *overflow* ou uma subtração resulta em *underflow*, o resultado é alterado, respectivamente, para o maior ou o menor valor representável. No exemplo precedente, temos, com saturação aritmética, o seguinte resultado:

```
 F000h = 1111 0000 0000 0000
+3000h = 0011 0000 0000 0000
        10010 0000 0000 0000
         1111 1111 1111 1111 = FFFFh
```

Para dar uma idéia do uso de instruções MMX, considere o seguinte exemplo, tirado de Peleg (1997) e Intel (1998b). Uma aplicação comum de vídeo é o efeito de *fade out* e *fade in*, no qual uma cena gradualmente se dissolve em outra. Duas imagens, A e B, são combinadas com uma média ponderada:

```
Pixel_resultado = Pixel_A × fade + Pixel_B × (1 - fade)
```

Esse cálculo é efetuado em cada posição de pixel de A e B. Se for produzida uma série de quadros de vídeo, variando gradualmente o valor do *fade* de 1 a 0 (em graduações adequadas para um número inteiro de 8 bits), a imagem resultante será uma passagem gradual da imagem A para a B.

A Figura 9.10 mostra a seqüência de passos requerida para um conjunto de pixels. Os pixels de 8 bits são convertidos em elementos de 16 bits, para acomodar as capacidades de multiplicação em 16 bits do MMX. Se as imagens usam uma resolução de 640 x 480 e a técnica de dissolver imagens usa todos os 255 possíveis valores de *fade*, o número total de instruções executadas, usando MMX, será de 535 milhões. O mesmo cálculo, efetuado sem usar instruções MMX, requer 1,4 bilhão de instruções (Intel, 1998b).

Tipos de operações do PowerPC

O PowerPC oferece uma grande coleção de tipos de operações. A Tabela 9.11 mostra esses tipos e apresenta exemplos de cada um deles. Diversas características merecem ser notadas.

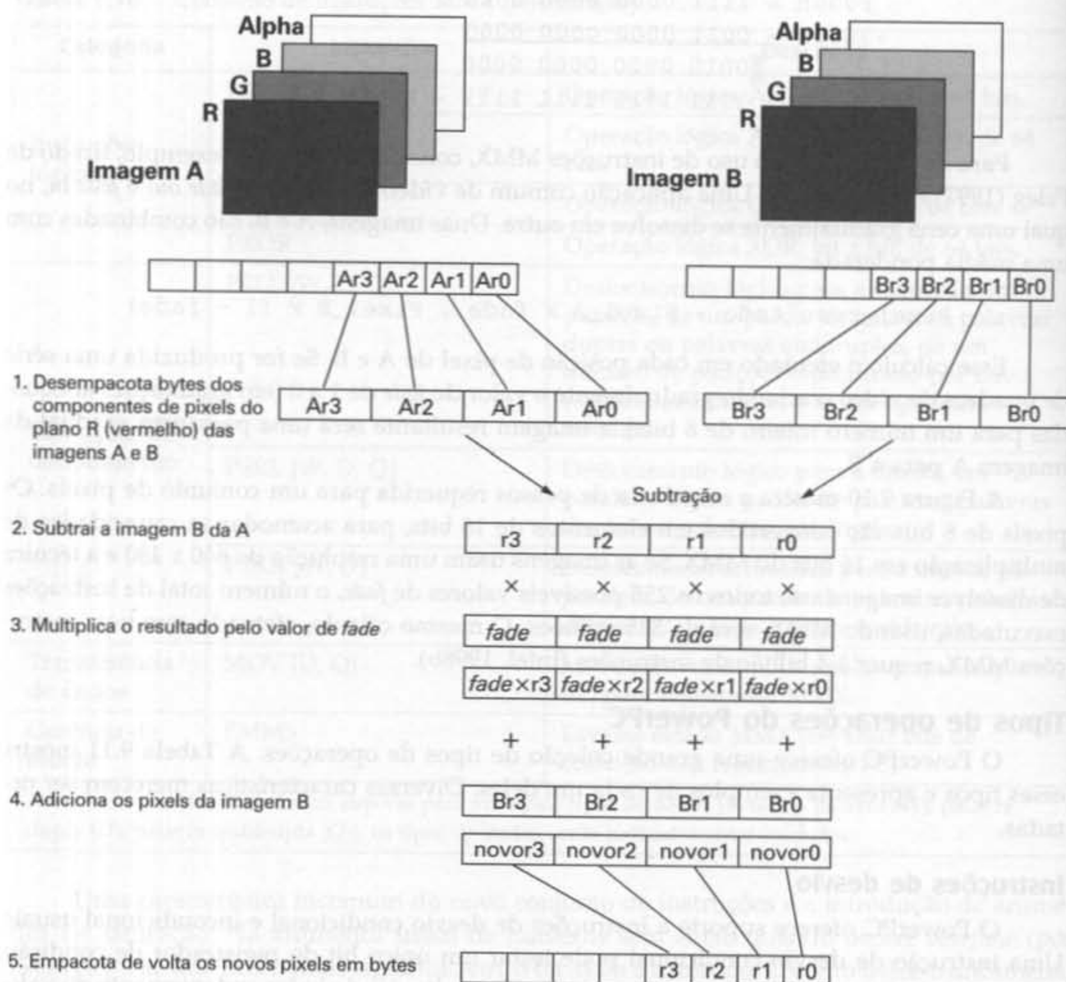
Instruções de desvio

O PowerPC oferece suporte a instruções de desvio condicional e incondicional usuais. Uma instrução de desvio condicional pode testar um único bit do registrador de condição, desviando se ele tem valor 0, ou se ele tem valor 1, ou desviando independentemente do valor desse bit. Pode também testar o conteúdo do registrador contador, desviando se seu valor é zero, ou se é diferente de zero, ou independentemente do seu valor. Portanto, nove condições distintas podem ser especificadas em uma instrução de desvio condicional. No teste de contador igual a zero ou diferente de zero, o valor do contador é decrementado de 1 antes do teste. Essa operação é conveniente para controlar laços de repetição.

As instruções de desvio podem também especificar que o endereço seguinte ao endereço de desvio deve ser armazenado no registrador de ligação, descrito no Capítulo 13. Isso facilita o processamento de chamada e retorno de procedimentos.

Instruções de carga e armazenamento

Na arquitetura PowerPC, apenas as instruções de carga e armazenamento referenciam posições de memória. As instruções aritméticas e lógicas são realizadas somente com registradores. Essa é uma característica de projetos RISC e será discutida mais adiante, no Capítulo 12.



Seqüência de código MMX para efetuar essa operação:

```
pxor        mm7, mm7           ;atribui valor zero a mm7
movq        mm3, valor_fade    ;carrega o valor de fade replicado 4 vezes
movd        mm0, imagemA       ;carrega 4 componentes de pixel vermelho da imagem A
movd        mm1, imagemB       ;carrega 4 componentes de pixel vermelho da imagem B
punpckblw   mm0, mm7           ;descomprime 4 pixels para 16 bits
punpckblw   mm1, mm7           ;descomprime 4 pixels para 16 bits
psubw       mm0, mm1           ;subtrai a imagem B da imagem A
pmulhw      mm0, mm3           ;multiplica o resultado da subtração pelo valor de fade
padddw      mm0, mm1           ;adiciona o resultado à imagem B
packuswb    mm0, mm7           ;empacota de volta os resultados de 16 bits em bytes
```

Figura 9.10 Combinação de imagens em representação de plano de cores (Peleg, 1997).

Tabela 9.11 Tipos de operações do PowerPC (com exemplos de operações típicas)

Instrução	Descrição
	Instruções de desvio
b	Desvio incondicional
bl	Desvia para o endereço-alvo e coloca o endereço efetivo da instrução seguinte ao desvio no Registrador de Ligação
bc	Desvio condicional baseado no conteúdo do Registrador Contador e/ou no valor de um bit do Registrador de Condição
sc	Chamada ao sistema para solicitar um serviço de sistema operacional
trap	Compara dois operandos e chama a rotina de tratamento de interrupções do sistema se as condições especificadas forem satisfeitas
	Instruções de carga e armazenamento
lwzu	Carrega palavra e estende com zeros para a esquerda; atualiza o registrador fonte
ld	Carrega palavra dupla
lmw	Carrega múltiplas palavras; carrega palavras consecutivas em registradores contíguos, a partir do registrador-alvo, até o registrador de propósito geral de número 31
lswx	Carrega uma seqüência de bytes nos registradores, com 4 bytes por registrador; começa pelo registrador-alvo e volta para o registrador 0 depois do registrador 31
	Instruções de aritmética inteira
add	Soma os conteúdos de dois registradores e coloca o resultado no terceiro registrador
sub	Subtrai os conteúdos de dois registradores e coloca o resultado no terceiro registrador
mullw	Multiplica os conteúdos dos 32 bits de mais baixa ordem de dois registradores e coloca o produto de 64 bits no terceiro registrador
divd	Divide o conteúdo de dois registradores de 64 bits e coloca o quociente no terceiro registrador
	Operações lógicas e de deslocamento
cmp	Compara dois operandos e atualiza quatro bits de condição no campo especificado do registrador de condição
crand	Efetua a operação lógica AND de dois bits do Registrador de Condição e coloca o resultado em uma dessas duas posições de bit
and	Efetua a operação lógica AND dos conteúdos de dois registradores e coloca o resultado no terceiro registrador
cntlzd	Conta o número de bits consecutivos iguais a 0, a partir do bit zero do registrador fonte, e coloca o resultado no registrador destino
rldic	Efetua a rotação para a esquerda de um registrador de palavra dupla, seguida por AND do conteúdo desse registrador com uma máscara, e armazena o resultado no registrador destino
sld	Desloca os bits do registrador fonte para a esquerda e coloca o resultado no registrador destino

(*continua*)

Tabela 9.11 Tipos de operações do PowerPC (com exemplos de operações típicas) (*continuação*)

Instrução	Descrição
	Instruções de ponto flutuante
lfs	Carrega o número de ponto flutuante de 32 bits da memória, converte para o formato de 64 bits e armazena em um registrador de ponto flutuante
fadd	Adiciona os conteúdos de dois registradores e coloca o resultado no terceiro registrador
fmadd	Multiplica os conteúdos de dois registradores, adiciona esse valor a um terceiro registrador e coloca o resultado no quarto registrador
fcmpu	Compara dois operandos de ponto flutuante e atualiza os bits de condição
	Instruções de gerenciamento de cache
dcbf	Pesquisa a memória cache de dados por um determinado endereço e executa uma operação de esvaziamento (*flushing*)
icbi	Invalida o bloco da memória cache de instruções

Dois aspectos caracterizam as diferentes instruções de carga e armazenamento:

- **Tamanho dos dados:** os dados podem ser transferidos em unidades de byte, meia palavra, palavra ou palavra dupla. Existem também instruções para carregar ou armazenar uma seqüência de bytes em/de múltiplos registradores.
- **Extensão de sinal:** ao carregar meia palavra ou uma palavra, os bits mais à esquerda do registrador destino de 64 bits podem ser preenchidos com zeros ou com o bit de sinal do valor carregado.

9.6 LINGUAGEM DE MONTAGEM

A CPU é capaz de entender e executar instruções de máquina. Essas instruções são constituídas, simplesmente, de números binários armazenados no computador. Se um programador quisesse programar diretamente em linguagem de máquina, teria de escrever um programa na forma de dados binários.

Considere um comando simples, tal como:

```
N = I + J + K
```

Suponha que desejamos programar esse comando em linguagem de máquina, inicializando I, J e K com os valores 2, 3 e 4, respectivamente. Isso é mostrado na Figura 9.11a. O programa é carregado a partir da posição de memória de endereço 101 (hexadecimal). É reservada uma área de memória para as quatro variáveis, a partir do endereço 201. O programa consiste em quatro instruções:

1. Carrega o conteúdo da posição de endereço 201 em AC.
2. Adiciona o conteúdo da posição de endereço 202 a AC.
3. Adiciona o conteúdo da posição de endereço 203 a AC.
4. Armazena o conteúdo de AC na posição de endereço 204.

Programar dessa maneira é, claramente, um processo tedioso e propenso a erros.

Uma ligeira melhora pode ser obtida ao escrever o programa em notação hexadecimal, em lugar da notação binária (Figura 9.11b). O programa pode ser escrito como uma série de linhas, cada qual com o endereço de uma posição de memória e o código hexadecimal do valor binário a ser armazenado naquela posição. É necessário, então, um programa que aceite essa entrada, traduza cada linha do programa como um número binário e armazene-o na posição de memória especificada.

Para facilitar um pouco mais, podemos usar um nome simbólico, ou mnemônico, para cada instrução. Como resultado, obtemos o *programa simbólico* mostrado na Figura 9.11c. Cada linha do programa consiste em três campos, separados por espaços. O primeiro campo contém o endereço de uma posição de memória. O segundo contém um símbolo de três letras para o código da operação. Se a instrução for uma instrução de referência à memória, o terceiro campo conterá o endereço referenciado. Para armazenar um dado arbitrário em uma determinada posição, inventamos uma *pseudo-instrução*, representada pelo símbolo DAT. Ela indica que o terceiro campo da linha contém um número hexadecimal, a ser armazenado na posição especificada pelo primeiro campo.

Para aceitar esse tipo de entrada, precisamos de um programa ligeiramente mais complexo. Esse programa lê cada linha da entrada, gera um número binário, com base nos valores do segundo e do terceiro campos (se houver), e armazena esse número na posição de memória especificada pelo primeiro campo.

O uso de um programa simbólico torna a tarefa de programação mais fácil, mas ainda é ineficaz. Em particular, temos de fornecer um endereço absoluto para cada palavra. Isso significa que o programa e os dados somente podem ser carregados em uma dada posição da memória, conhecida antes da carga do programa. Pior que isso, se algum dia quisermos alterar o programa, acrescentando ou apagando alguma linha, os endereços de todas as palavras subseqüentes terão de ser alterados.

Um mecanismo muito melhor, e normalmente empregado, é usar endereços simbólicos. Isso é mostrado na Figura 9.11d. Cada linha é constituída ainda de três campos. O primeiro campo é um endereço, mas um símbolo é usado no lugar de um endereço numérico absoluto. Algumas linhas não têm endereço, o que significa que o endereço dessa linha é o endereço consecutivo ao da linha anterior. Em instruções de referência à memória, o terceiro campo contém também um endereço simbólico.

Com esse último refinamento, temos uma *linguagem de montagem*. Programas escritos em linguagem de montagem são traduzidos para linguagem de máquina por um *montador*. Esse programa não apenas deve fazer a tradução simbólica discutida anteriormente mas também associar um endereço de memória a cada endereço simbólico.

O desenvolvimento de linguagens de montagem constituiu um grande marco na evolução da tecnologia de computadores. Ele foi o primeiro passo para o desenvolvimento das linguagens de alto nível usadas atualmente. Embora poucos programadores usem a linguagem de montagem, quase todas as máquinas têm uma. Elas são utilizadas por programas de sistema, tais como compiladores e rotinas de E/S.

Endereço	Conteúdo			
101	0010	0010	0000	0001
102	0001	0010	0000	0010
103	0001	0010	0000	0011
104	0011	0010	0000	0100
201	0000	0000	0000	0010
202	0000	0000	0000	0011
203	0000	0000	0000	0100
204	0000	0000	0000	0000

(a) Programa binário

101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Programa simbólico

Endereço	Conteúdo
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Programa hexadecimal

Rótulo	Operação	Operando
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Programa em linguagem de montagem

Figura 9.11 Computação da fórmula N = I + J + K.

9.7 LEITURA RECOMENDADA

Diversos livros oferecem uma boa abordagem sobre linguagens de máquina e projeto de conjunto de instruções, incluindo Hennessy (1996), Tanenbaum (1990) e Hayes (1988). O conjunto de instruções do Pentium é descrito em Brey (1997), e Dewar e Smosna (1990). O conjunto de instruções do PowerPC é descrito em IBM (1994) e Weiss (1994).

9.8 EXERCÍCIOS

9.1 Muitas CPUs incluem lógica para efetuar operações aritméticas sobre números decimais empacotados. Embora as regras da aritmética decimal sejam semelhantes às usadas nas operações sobre números binários, se for usada lógica binária, os resultados decimais podem requerer algumas correções em dígitos individuais.
Considere a adição decimal de dois números sem sinal. Se cada número é constituído de N dígitos, então cada número tem $4N$ bits. Dois números devem ser adicionados usando um somador binário. Sugira uma regra simples para corrigir o resultado. Efetue a adição dos números 1698 e 1786 usando essa regra de correção.

9.2 O complemento de dez de um número decimal X é definido como $10N - X$, onde N é o número de dígitos decimais do número. Descreva o uso da representação em complemento de dez para efetuar a subtração decimal. Descreva esse procedimento, subtraindo $(0326)_{10}$ de $(0736)_{10}$.

9.3 Compare máquinas com instruções de zero, um, dois e três endereços, escrevendo um programa, em cada uma dessas quatro máquinas, para implementar o comando:

```
X = (A + B × C)/(D - E × F)
```

As instruções disponíveis para as quatro máquinas são as seguintes:

0 endereço	1 endereço	2 endereços	3 endereços
PUSH M	LOAD M	MOVE X, Y ($X \leftarrow Y$)	MOVE X, Y ($X \leftarrow Y$)
POP M	STORE M	ADD X, Y ($X \leftarrow X + Y$)	ADD X, Y, Z ($X \leftarrow Y + Z$)
ADD	ADD M	SUB X, Y ($X \leftarrow X - Y$)	SUB X, Y, Z ($X \leftarrow Y - Z$)
SUB		MUL X, Y ($X \leftarrow X \times Y$)	MUL X, Y, Z ($X \leftarrow Y \times Z$)
MUL	MUL M		DIV X, Y, Z ($X \leftarrow Y/Z$)
DIV	DIV M	DIV X, Y ($X \leftarrow X/Y$)	

9.4 Considere um computador hipotético com um conjunto de instruções com apenas duas instruções de n bits. O primeiro bit especifica o código de operação e os bits restantes especificam uma das $2^n - 1$ palavras de n bits da memória principal. As duas instruções são:

SUBS X Subtrai o conteúdo da posição de memória de endereço X do acumulador e armazena o resultado na posição X e no acumulador.

JUMP X Coloca o endereço X no contador de programa.

Uma palavra da memória principal pode conter uma instrução ou um número binário, em notação de complemento de dois. Mostre que esse repertório de instruções é razoavelmente completo, especificando como as seguintes operações podem ser programadas:

a. Transferência de dados: da posição X para o acumulador, do acumulador para a posição X
b. Adição: somar o conteúdo da posição X ao acumulador
c. Desvio condicional
d. Operação lógica OU
e. Operações de E/S

9.5 Muitos conjuntos de instruções contêm uma instrução NOOP, que significa nenhuma operação e que não tem qualquer efeito sobre o estado da CPU, exceto incrementar o contador de programa. Sugira alguns usos dessa instrução.

9.6 Suponha que uma CPU utilize uma pilha para gerenciar a chamada e o retorno de procedimentos. Seria possível eliminar o contador de programa, usando o topo da pilha como contador de programa?

9.7 No Apêndice 9A, dizemos que um conjunto de instruções não inclui operações explícitas sobre a pilha, se a pilha for usada pela CPU apenas com o propósito de manipular procedimentos. Como a CPU poderia usar a pilha para qualquer propósito, sem que o conjunto de instruções incluísse operações sobre pilha?

9.8 Converta as seguintes fórmulas, da notação pós-fixa para a notação infixada.
a. AB + C + D*
b. AB/CD/ +
c. ABCDE + × × /
d. ABCDE + F/ + G − H/ × +

9.9 Converta as seguintes fórmulas, da notação infixada para a notação pós-fixa.
a. A + B + C + D + E
b. (A + B) × (C + D) + E

c. (A × B) + (C × D) + E

d. (A – B) × (((C – D × E)/F)/G) × H

9.10 Converta a expressão $A + B - C$ para a notação pós-fixa, usando o algoritmo de Dijkstra. Mostre os passos envolvidos. O resultado é equivalente a $(A + B) - C$ ou a $A + (B - C)$? Essa diferença é importante?

9.11 A arquitetura do Pentium II inclui uma instrução chamada Ajuste Decimal após Adição (DAA). A instrução DAA efetua a seguinte seqüência de operações:

```
if ((AL AND 0FH) > 9) OR (AF = 1) then
   AL ← AL + 6;
   AF ← 1;
else
   AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1) then
   AL ← AL + 60H;
   CF ← 1;
else
   AF ← 0;
endif.
```

'H' indica hexadecimal. AL é um registrador de 8 bits que mantém o resultado da adição de dois números inteiros de 8 bits, sem sinal. AF é um bit de condição que é atualizado com valor 1 se ocorrer 'vai-um' do bit 3 para o bit 4, no resultado de uma adição. CF é um bit de condição que é atualizado com valor 1 se ocorrer 'vai-um' do bit 7 para o bit 8. Explique a função computada pela instrução DAA.

9.12 A instrução de comparação do Pentium II (CMP) subtrai o operando fonte do operando destino e atualiza os bits de condição (C, P, A, Z, S, O) do registrador de estado, mas não altera nenhum dos operandos. A instrução CMP pode ser seguida de um desvio condicional (Jcc) ou de uma instrução que atualiza os bits de condição (SETcc), onde cc designa uma das 16 condições apresentadas na Tabela 9.9. Mostre que as condições testadas na comparação de números com sinal são corretas.

9.13 A maioria dos conjuntos de instruções de microprocessadores inclui uma instrução para testar uma condição e atualizar o valor do operando de destino se a condição for verdadeira. Alguns exemplos são as instruções SETcc do Pentium II, Scc do Motorola MC68000 e Scond do National NS32000.

a. Existem algumas diferenças entre essas instruções:

- SETcc e Scc operam apenas sobre um byte, enquanto Scond opera sobre operandos com tamanho de byte, palavra ou palavra dupla.
- SETcc e Scond atualizarão o operando de destino com o valor inteiro 1, se a condição for verdadeira, e com zero, se for falsa. Scc preencherá todos os bits do byte de destino com 1, se a condição for verdadeira, e com todos os bits 0, se for falsa.

Quais as vantagens e desvantagens relativas dessas diferenças?

b. Nenhuma dessas instruções altera qualquer bit de condição; portanto, é necessário um teste explícito para determinar o valor do resultado da instrução. Discuta se seria conveniente atualizar os códigos de condição como resultado dessa instrução.

c. Um simples comando **IF**, tal como **IF** a > b **THEN**, pode ser implementado usando um método de representação numérica (isto é, tornando explícito o valor booleano) ou usando o método do *fluxo de controle*, que representa o valor de uma expressão booleana por um certo ponto no programa. Um compilador pode implementar o comando **IF** a > b **THEN** pelo seguinte código do 80x86:

```
        SUB     CX,CX     ;atualiza o registrador CX com o valor 0
        MOV     AX, B     ;move o conteúdo da posição de memória B para o
                          registrador AX
        CMP     AX, A     ;compara o conteúdo do registrador AX  com o da
                          posição A
        JLE     TEST      ;desvia se A ≤ B
        INC     CX        ;incrementa o conteúdo do registrador CX
TEST    JCXZ    OUT       ;desvia se o conteúdo de CX é igual a 0
THEN

OUT
```

O resultado de (A > B) é um valor booleano, mantido em um registrador e disponível, mais tarde, fora do contexto do fluxo de código mostrado. É conveniente usar, para isso, o registrador CX, porque muitas operações de desvio e operações de controle de laços de repetição incluem, implicitamente, um teste do valor do registrador CX.

Mostre uma implementação alternativa usando a instrução SETcc, que economize memória e tempo de execução. (*Dica:* não são necessárias outras instruções 80x86 adicionais, além da instrução SETcc.)

d. Considere agora o seguinte comando de uma linguagem de alto nível:

```
A : = (B > C) OR (D = F)
```

Um compilador pode gerar o seguinte código:

```
     MOV   EAX, B    ;move o conteúdo da posição de memória B para o
                     registrador EAX
     CMP   EAX, C    ;compara o conteúdo do registrador EAX com o da
                     posição C
     MOV   BL, 0     ;0 representa falso
     JLE   N1        ;desvia se B ≤ C
     MOV   BL, 1     ;1 representa falso
N1   MOV   EAX, D    ;move o conteúdo da posição D para o registrador EAX
     CMP   EAX, F    ;compara o conteúdo do registrador EAX com
                     ;o da posição F
     MOV   BH, 0     ;0 representa falso
     JNE   N2        ;desvia se D ≠ F
     MOV   BH, 1     ;1 representa verdadeiro
N2   OR    BL, BH    ;operação OR
```

Mostre uma implementação alternativa usando a instrução SETcc, que economize memória e tempo de execução.

9.14 Usando o algoritmo de conversão de notação infixada para a notação pós-fixa, definido no Apêndice 9A, mostre os passos envolvidos na conversão da expressão da Figura 9.15 para a notação pós-fixa. Use uma apresentação semelhante à da Figura 9.17.

9.15 Mostre como é feito o cálculo da expressão da Figura 9.17, usando uma apresentação semelhante à da Figura 9.16.

9.16 Refaça o leiaute da disposição de bytes little-endian mostrado na Figura 9.18, de maneira que os bytes sejam numerados conforme a disposição big-endian. Ou seja, desenhe a memória como sendo composta por linhas de 64 bits, com os bytes listados da esquerda para a direita e de cima para baixo.

9.17 Represente as seguintes estruturas de dados, para as disposições de bytes big-endian e little-endian, usando o formato da Figura 9.18. Comente os resultados.

a.
```
struct {
    double   i;  //0x1112131415161718
} s1;
```
b.
```
struct {
     int   i;    //0x11121314
     int   j;    //0x15161718
} s2;
```
c.
```
struct {
     short  i;  //0x1112
     short  j;  //0x1314
     short  k;  //0x1516
     short  l;  //0x1718
} s3;
```

9.18 A especificação da arquitetura do PowerPC não diz como o processador deve implementar a disposição de bytes little-endian, indicando apenas como o processador deve ver a memória. Para converter uma estrutura de dados da disposição big-endian para a little-endian, o processador pode implementar um mecanismo de troca de bytes ou usar algum tipo de mecanismo de tradução de endereços. Todos os processadores PowerPC atuais usam a disposição big-endian como padrão e utilizam um mecanismo de tradução de endereços para tratar dados com a disposição little-endian.
Considere a estrutura *s* definida na Figura 9.18. O leiaute na parte inferior direita da figura mostra a estrutura *s*, tal como é vista pelo processador. De fato, se a estrutura *s* for compilada com a disposição little-endian, sua representação na memória é tal como mostrado na Figura 9.12. Explique o mapeamento envolvido, descreva uma maneira fácil de implementar esse mapeamento e discuta a eficácia dessa abordagem.

9.19 Escreva um pequeno programa para determinar o tipo de disposição de bytes usado em uma máquina. Execute seu programa em um computador disponível e verifique a saída.

Mapeamento de endereços little-endian

Endereço de byte								
					11	**12**	**13**	**14**
00	00	01	02	03	04	05	06	07
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**
08	08	09	0A	0B	0C	0D	0E	0F
	'D'	**'C'**	**'B'**	**'A'**	**31**	**32**	**33**	**34**
10	10	11	12	13	14	15	16	17
			51	**52**		**'G'**	**'F'**	**'E'**
18	18	19	1A	1B	1C	1D	1E	1F
					61	**62**	**63**	**64**
20	20	21	22	23	24	25	26	27

Figura 9.12 Leiaute da memória da estrutura *s* com disposição little-endian no PowerPC.

APÊNDICE 9A PILHAS

Pilhas

Uma *pilha* é um conjunto ordenado de elementos, dos quais apenas um pode ser acessado em um dado instante. Esse elemento é denominado *topo* da pilha. O número de elementos da pilha, ou *tamanho* da pilha, é variável. Os itens só podem ser adicionados ou removidos a partir do topo da pilha. Por essa razão, a pilha é também conhecida como *lista pushdown* ou *lista LIFO (last-in-first-out)*.

A Figura 9.13 mostra as operações básicas sobre pilhas. Inicialmente, consideramos que a pilha contém certo número de elementos. A operação PUSH coloca um novo item no topo da pilha. A operação POP remove o item que está no topo da pilha. Nessas duas operações, o topo da pilha se move adequadamente. Operações binárias, que requerem dois operandos (por exemplo, multiplicação, divisão, soma, subtração), retiram os dois operandos do topo da pilha e colocam o resultado de volta na pilha. Operações unárias, que requerem apenas um operando (por exemplo, a operação lógica NOT), usam o item do topo da pilha. Essas operações são resumidas na Tabela 9.12.

Tabela 9.12 Operações sobre pilhas

PUSH	Coloca um novo elemento no topo da pilha.
POP	Retira o elemento do topo da pilha.
Operação unária	Efetua a operação sobre o elemento do topo da pilha. Substitui o elemento do topo pelo resultado.
Operação binária	Efetua a operação sobre os dois elementos no topo da pilha. Retira os dois elementos do topo da pilha. Coloca o resultado da operação no topo da pilha.

Implementação da pilha

É útil fornecer uma pilha como parte da implementação de uma CPU. Como foi discutido na Seção 9.4, a pilha pode ser usada para gerenciar a chamada e o retorno de procedi-

mentos. Ela também pode ser útil para o programador. Um exemplo é a avaliação de expressões, discutida no decorrer desta seção.

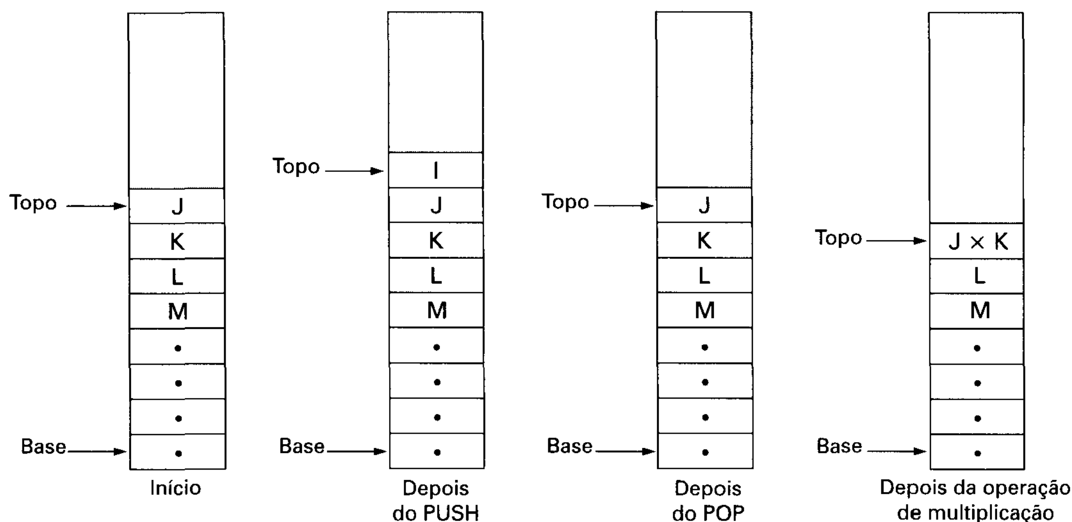


Figura 9.13 Operação básica sobre uma pilha.

A implementação da pilha depende, em parte, do uso ao qual ela se destina. Se quisermos tornar as operações sobre a pilha disponíveis para o programador, então, o conjunto de instruções deverá incluir instruções orientadas a pilha, incluindo PUSH, POP e operações que usam um ou dois elementos no topo da pilha como operandos. Como essas operações referenciam uma única posição de memória, ou seja, o topo da pilha, o endereço do operando ou operandos não precisa ser incluído na instrução, ficando implícito. Essas são as instruções de zero endereços mencionadas na Seção 9.1.

Se o mecanismo de pilha for usado apenas pela CPU, por exemplo para manipular procedimentos, então o conjunto de instruções não deverá incluir nenhuma instrução que opere sobre a pilha de maneira explícita. Em ambos os casos, a implementação da pilha requer um conjunto de posições de memória para armazenar os elementos da pilha. Uma abordagem típica é mostrada na Figura 9.14a. Um bloco contíguo de posições de memória principal (ou memória virtual) é reservado para a pilha. Na maior parte do tempo, esse bloco de memória fica parcialmente preenchido com elementos da pilha, ficando o restante disponível para o crescimento da pilha.

Para manipular a pilha de maneira adequada, é necessário manter três endereços, que são freqüentemente armazenados em registradores da CPU:

- **Apontador de topo da pilha:** contém o endereço do topo da pilha. Se um item for colocado ou retirado da pilha, o apontador de topo da pilha será incrementado ou decrementado, respectivamente, de modo que contenha o endereço do novo topo da pilha.
- **Base da pilha:** contém o endereço da posição inicial do bloco de memória reservado para a pilha. Se ocorrer uma tentativa de executar a operação POP (desempilhar) quando a pilha estiver vazia, será reportado um erro.

- **Limite da pilha:** contém o endereço da outra extremidade do bloco de memória reservado para a pilha. Se ocorrer uma tentativa de executar a operação PUSH (empilhar) quando o bloco estiver totalmente utilizado pela pilha, será reportado um erro.

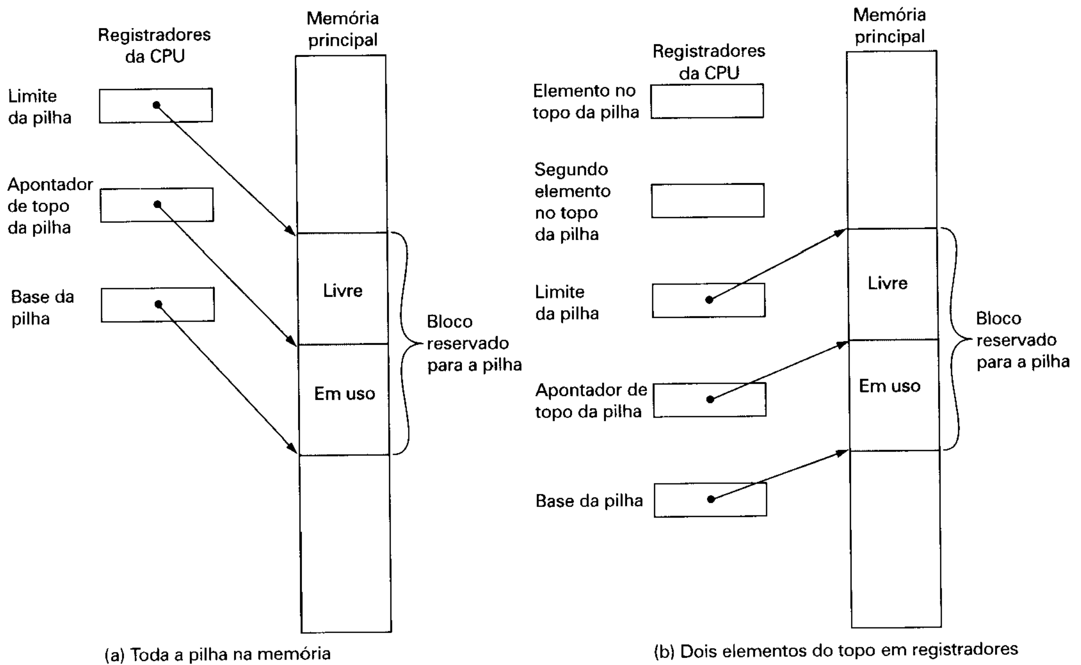


Figura 9.14 Organizações típicas de pilha.

Tradicionalmente e na maioria das máquinas atuais, a base da pilha fica no endereço mais alto do bloco reservado para a pilha e o limite situa-se no endereço mais baixo. Assim, a pilha cresce dos endereços altos para os endereços baixos.

Para acelerar as operações de pilha, os dois elementos do topo são freqüentemente armazenados em registradores, como mostrado na Figura 9.14b. Nesse caso, o apontador de topo da pilha contém o endereço do terceiro elemento da pilha.

Avaliação de expressões

Fórmulas matemáticas normalmente são expressas na notação infixa. Nessa notação, um operador binário é utilizado entre os operandos (por exemplo, a + b). Parênteses são usados, em expressões mais complexas, para determinar a ordem de avaliação de expressões. Por exemplo, a expressão a + (b × c) tem um resultado diferente de (a + b) × c. Usualmente, estabelecemos precedências entre os operadores, para minimizar o uso de parênteses. A multiplicação geralmente tem precedência sobre adição; assim, a + b × c é equivalente a a + (b × c).

Uma técnica alternativa é o uso de notação pós-fixa ou polonesa reversa. Nessa notação, o operador vem depois de seus operandos. Por exemplo,

a + b	se torna a b +
a + (b × c)	se torna a b c × +
(a + b) × c	se torna a b + c ×

Observe que a notação pós-fixa não requer parênteses, independentemente da complexidade da expressão.

A vantagem dessa notação é que uma expressão nessa forma é facilmente avaliada usando uma pilha. A expressão em notação pós-fixa é lida da esquerda para a direita e, para cada elemento da expressão, são aplicadas as seguintes regras:

1. Se o elemento for uma variável ou constante, coloque-o no topo da pilha.
2. Se o elemento for um operador, retire os operandos do topo da pilha, efetue a operação e empilhe o resultado.

Depois que toda a expressão for processada, o resultado estará no topo da pilha.

A simplicidade desse algoritmo o torna conveniente para avaliar expressões. Por isso, muitos compiladores traduzem expressões da maneira infixa, tal como ocorrem em linguagens de alto nível, para a forma pós-fixa, e geram instruções de máquina a partir dessa notação. A Figura 9.15 mostra a seqüência de instruções de máquina para avaliar o comando f = (a – b)/(c + d × e), usando instruções de pilha. A figura apresenta também a implementação desse comando ao usar instruções de um endereço e de dois endereços. Note que, mesmo não sendo usadas instruções de pilha nesses dois últimos casos, a notação pós-fixa serve para guiar a geração de instruções de máquina. A seqüência de eventos para a utilização da pilha na computação da expressão é mostrada na Figura 9.16.

	Pilha	Registradores de propósito geral	Registrador único
	Push a Push b Subtract Push c Push d Push e Multiply Add Divide Pop f	Load G[1], a Subtract G[1], b Load G[2], d Multiply G[2], e Add G[2], c Divide G[1], G[2] Store G[1], f	Load d Multiply e Add c Store f Load a Subtract b Divide f Store f
Número de instruções	10	7	8
Referências à memória	10 op + 6 d	7 op + 6 d	8 op + 8 d

Figura 9.15 Comparação de três programas para calcular f = (a – b)/(c + d × e).

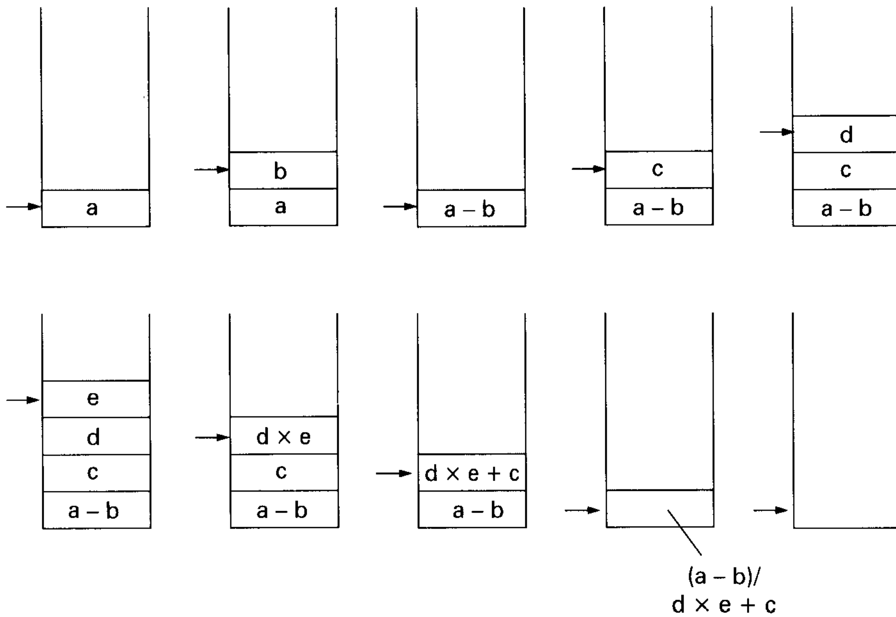


Figura 9.16 Uso de pilha para computar f = (a – b)/(d × e + c).

Entrada	Saída	Pilha (topo à direita)
A + B × C + (D + E) × F	vazia	vazia
+ B × C + (D + E) × F	A	vazia
B × C + (D + E) × F	A	+
× C + (D + E) × F	A B	+
C + (D + E) × F	A B	+ ×
+ (D + E) × F	A B C	+ ×
(D + E) × F	A B C × +	+
D + E) × F	A B C × +	+ (
+ E) × F	A B C × + D	+ (
E) × F	A B C × + D	+ (+
) × F	A B C × + D E	+ (+
× F	A B C × + D E +	+
F	A B C × + D E +	+ ×
vazia	A B C × + D E + F	+ ×
vazia	A B C × + D E + F × +	vazia

Figura 9.17 Conversão de uma expressão na notação infixa para pós-fixa.

O próprio processo de converter uma expressão infixa para a notação pós-fixa é efetuado mais facilmente usando uma pilha. O seguinte algoritmo foi proposto por Dijkstra (1963). A expressão infixa é lida da esquerda para a direita e é produzida a expressão pós-fixa correspondente, por meio dos seguintes passos:

1. Examine o próximo elemento na entrada.
2. Se for um operando, copie-o na saída.
3. Se for um abre parênteses, coloque-o no topo da pilha.
4. Se for um operador, então:
   - Se o topo da pilha for um abre parênteses, empilhe o operador.
   - Se esse operador tiver maior precedência que o operador no topo da pilha (a multiplicação e a divisão têm maior precedência que a adição e a subtração), então empilhe o operador.
   - Senão, retire a operação do topo da pilha, copiando-a na saída, e repita o passo 4.
5. Se for um fecha parênteses, retire os operadores do topo da pilha, copiando-os na saída, até que seja encontrado um abre parênteses. Retire o abre parênteses do topo da pilha e descarte-o.
6. Se ainda houver dados na entrada, volte ao passo 1.
7. Se não houver dados na entrada, desempilhe os operandos restantes.

A Figura 9.17 mostra o uso desse algoritmo. O exemplo deve dar a você uma idéia do poder dos algoritmos baseados em pilha.

APÊNDICE 9B LITTLE-ENDIAN, BIG-ENDIAN E BI-ENDIAN

Uma questão importante é o modo como bytes são representados e referenciados dentro de uma palavra ou como os bits são representados e referenciados dentro de um byte. Consideramos primeiramente o problema da disposição de bytes e, em seguida, o de bits.

Disposição de bytes

O conceito da disposição dos bytes (*endianness*) foi discutido pela primeira vez na literatura por Cohen (1981). O *endianness* diz respeito à disposição dos bytes de valores escalares de múltiplos bytes. Essa questão é mais facilmente apresentada por meio de um exemplo. Suponha que o valor hexadecimal 12345678 esteja armazenado em uma palavra de 32 bits, a partir da posição de endereço de byte 184, em uma memória endereçável em unidades de byte. Esse valor é constituído de 4 bytes, com o byte menos significativo contendo o valor 78 e o byte mais significativo contendo o valor 12. Existem duas maneiras de armazenar esse valor:

Endereço	Valor
184	12
185	34
186	56
187	78

Endereço	Valor
184	78
185	56
186	34
187	12

No mapeamento mostrado à esquerda, o byte mais significativo é armazenado no menor endereço de byte; isso é conhecido como big-endian e é equivalente à ordem usual de escrita em linguagens da cultura ocidental. No mapeamento mostrado à direita, o byte menos significativo é armazenado no menor endereço de byte; isso é conhecido como little-endian e é remanescente da ordem de escrita de operações aritméticas sobre unidades aritméticas, da

direita para a esquerda[2]. Para um dado valor escalar de múltiplos bytes, o big-endian e o little-endian constituem mapeamentos de byte inversos um do outro.

O conceito de *endianness* surge quando é necessário tratar uma entidade de múltiplos bytes como um único item de dados, com um endereço único, embora seja composto de unidades endereçáveis menores. Algumas máquinas, como Intel 80x86, Pentium II, VAX e Alpha, são little-endian, enquanto outras, como IBM Sistema 370/390, Motorola 680 × 0, Sun SPARC e a maioria das máquinas RISC, usam o mapeamento big-endian. Isso apresenta problemas na transferência de dados entre máquinas com tipos de mapeamentos diferentes ou quando um programador tem de manipular bytes ou bits individuais em um valor escalar de múltiplos bytes.

A propriedade de *endianness* não se estende além de uma unidade individual de dados. Em qualquer máquina, agregados como arquivos, estruturas de dados e vetores são compostos de múltiplas unidades de dados, cada uma com seu mapeamento. Portanto, a conversão de um bloco de memória de um tipo de mapeamento para outro requer conhecimento da estrutura de dados.

A Figura 9.18 mostra como o *endianness* determina o endereçamento e a ordem de bytes. A estrutura, em linguagem C, apresentada no alto da figura, contém certo número de tipos de dados. O leiaute de memória mostrado na parte inferior esquerda é o resultado da compilação dessa estrutura para uma máquina big-endian; o leiaute mostrado na parte inferior direita corresponde a uma máquina little-endian. Em cada caso, a memória é representada como uma série de linhas de 64 bits. No caso do big-endian, a memória é tipicamente disposta da esquerda para a direita e de cima para baixo, enquanto no little-endian, a memória é tipicamente disposta da direita para a esquerda e de cima para baixo. Note que essas disposições são arbitrárias. Qualquer esquema poderia usar uma disposição da esquerda para a direita ou da direita para a esquerda, em uma linha; isso é apenas uma questão de ilustração e não de mapeamento da memória. De fato, uma variedade de formas de ilustração pode ser encontrada dentro do mesmo manual de programação de uma determinada máquina.

Diversas observações podem ser feitas sobre essa estrutura de dados:

- Cada item de dado tem o mesmo endereço nos dois esquemas. Por exemplo, o endereço da palavra dupla com valor hexadecimal 2122232425262728 é 08.
- Dentro de um valor escalar de múltiplos bytes, a disposição de bytes no big-endian é o inverso da disposição de bytes no little-endian.
- O *endianness* não afeta a organização de itens de dado dentro de uma estrutura. Portanto, os quatro bytes da palavra *c* aparecem invertidos nas duas disposições, mas o vetor de sete caracteres *d* não é invertido. Assim, o endereço de cada elemento individual do vetor *d* é o mesmo nas duas estruturas.

2. Os termos *big-endian* e *little-endian* vêm do livro *Gulliver's Travels* (As Viagens de Gulliver) de Jonathan Swift, Parte 1, Capítulo 4.

```
struct{
  int       a;     //0x1112_1314                              word
  int       pad;   //
  double    b;     //0x2122_2324_2526_2728                    doubleword
  char*     C;     //0X3132_3334                              word
  char      d[7];  //'A'.'B', 'C', 'D', 'E', 'F', 'G'         byte array
  short     e;     //0x5152                                   halfword
  int       f;     //0x6161_6364                              word
} s;
```

Mapeamento de endereços big-endian

Endereço de byte								
	11	**12**	**13**	**14**				
00	00	01	02	03	04	05	06	07
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**
08	08	09	0A	0B	0C	0D	0E	0F
	31	**32**	**33**	**34**	**'A'**	**'B'**	**'C'**	**'D'**
10	10	11	12	13	14	15	16	17
	'E'	**'F'**	**'G'**		**51**	**52**		
18	18	19	1A	1B	1C	1D	1E	1F
	61	**62**	**63**	**64**				
20	20	21	22	23				

Mapeamento de endereços little-endian

								Endereço de byte
				11	**12**	**13**	**14**	
07	06	05	04	03	02	01	00	00
21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	
0F	0E	0D	0C	0B	0A	09	08	08
'D'	**'C'**	**'B'**	**'A'**	**31**	**32**	**33**	**34**	
17	16	15	14	13	12	11	10	10
		51	**52**		**'G'**	**'F'**	**'E'**	
1F	1E	1D	1C	1B	1A	19	18	18
				61	**62**	**63**	**64**	
				23	22	21	20	20

Figura 9.18 Exemplo de uma estrutura de dados em C e suas respectivas disposições de bytes (IBM, 1994).

Talvez o efeito da disposição de bytes fique mais claro se olharmos a memória como um agrupamento vertical de bytes, como mostrado na Figura 9.19.

Não existe consenso sobre qual tipo de disposição de bytes é mais vantajoso. Os seguintes pontos são favoráveis ao big-endian:

- **Ordenação de seqüência de caracteres:** um processador big-endian é mais rápido na comparação de seqüências de caracteres alinhados em endereços de inteiros; a ULA pode comparar múltiplos bytes em paralelo.
- **Listagem de valores decimais ASCII:** todos os valores podem ser impressos da esquerda para a direita, sem causar confusão.
- **Ordem coerente:** processadores little-endian armazenam números inteiros e seqüências de caracteres na mesma ordem (os bytes mais significativos vêm primeiro).

Os seguintes pontos são favoráveis ao little-endian:

- Um processador big-endian tem de efetuar uma adição para converter um endereço de 32 bits para um endereço de 16 bits, a fim de obter os bytes menos significativos.
- É mais fácil efetuar a aritmética de alta precisão usando a disposição little-endian, pois não é necessário encontrar o byte menos significativo nem movê-lo para trás.

Essas diferenças são secundárias e a escolha do estilo de disposição de bytes é, quase sempre, apenas uma questão de manter compatibilidade com máquinas anteriores.

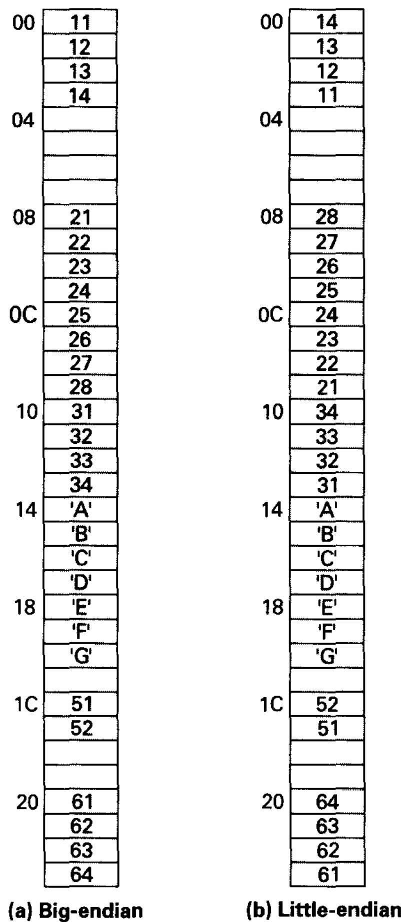


Figura 9.19 Outra visão da Figura 9.18.

O PowerPC é um processador bi-endian, ou seja, possibilita a utilização de ambas as disposições de bytes, big-endian e little-endian. A arquitetura big-endian permite aos desenvolvedores de software escolher qualquer uma das disposições de bytes, ao migrar sistemas operacionais ou aplicações de outras máquinas. O sistema operacional determina a disposição de bytes na qual um processo executa. Uma vez que a disposição é selecionada, todas as operações subseqüentes de carga e armazenamento na memória são determinadas pelo modelo de endereçamento de memória daquela disposição selecionada. Para oferecer suporte a essa característica de hardware, são usados 2 bits do registrador de estado da máquina (MSR), que é mantido pelo sistema operacional como parte do estado do processo. Um desses bits especifica a disposição de bytes na qual o núcleo do sistema operacional executa; o outro especifica a disposição do processo corrente. Portanto, é possível mudar a disposição de bytes para cada processo.

Disposição de bits

Ao dispor os bits dentro de um byte, vemo-nos diante de duas questões:

1. Os bits são numerados a partir de zero ou a partir de um?
2. O bit de menor ordem é o bit menos significativo do byte (little-endian) ou é o bit mais significativo do byte (big-endian)?

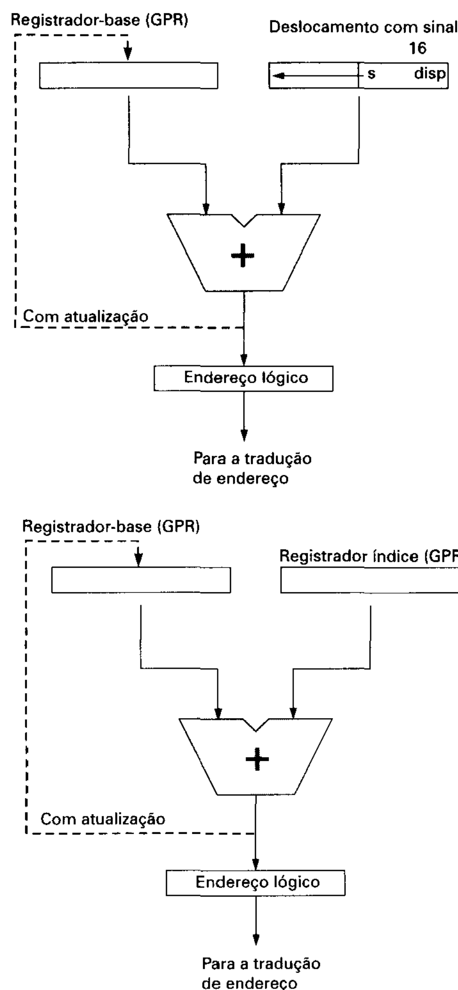
Essas questões não são respondidas da mesma maneira em todas as máquinas. De fato, em algumas máquinas, a resposta é diferente para diferentes circunstâncias. Além disso, a escolha da disposição dos bits dentro de um byte nem sempre é coerente com a disposição de bytes em um valor escalar de múltiplos bytes. O programador deve levar isso em conta ao manipular bits individuais.

Outra área em que esse aspecto deve ser considerado é a transmissão de dados por meio de uma linha serial. Ao transmitir cada byte individual, o sistema deve transmitir primeiro o bit mais significativo ou o bit menos significativo? O projetista deve certificar-se de que os bits recebidos sejam manipulados adequadamente. Para uma discussão sobre esse assunto, veja James (1990).

capítulo

10

Conjunto de Instruções: Modos de Endereçamento e Formatos



- Uma referência a um operando em uma instrução pode conter tanto o valor do operando (imediato) quanto o endereço do operando. Uma ampla variedade de modos de endereçamento é usada em diversos conjuntos de instruções. Isso inclui o endereçamento direto (o campo de endereço contém o endereço do operando), o endereçamento indireto (o campo de endereço aponta para uma posição de memória que contém o endereço do operando), de registrador indireto via registrador e várias formas de deslocamento, nas quais o valor de um registrador é adicionado a um endereço, para produzir o endereço efetivo do operando.
- O formato de uma instrução define a disposição dos campos de informação na instrução. O projeto de formatos de instrução é uma tarefa complexa, que inclui levar em consideração aspectos como o tamanho da instrução, que pode ser fixo ou variável, o número de bits reservados para o código de operação e para cada referência a operando e como o modo de endereçamento é determinado.

No Capítulo 9, focalizamos *o que* faz um conjunto de instruções. Em particular, examinamos os tipos de operação e de operandos que podem ser especificados em instruções de máquina. Este capítulo aborda a questão de *como* especificar operações e operandos nas instruções. Duas questões devem ser consideradas. A primeira é como o endereço de um operando é especificado e a segunda, como são organizados os bits de uma instrução, para definir a operação e os endereços de operandos da instrução.

10.1 ENDEREÇAMENTO

Em um formato de instrução típico, os campos de endereço são relativamente pequenos. Para possibilitar a referência a uma grande quantidade de posições de memória principal ou, em alguns sistemas, de memória virtual, várias técnicas de endereçamento têm sido empregadas. Todas essas técnicas envolvem decisões que contrapõem a quantidade de posições de memória endereçáveis e/ou a flexibilidade de endereçamento ao número de referências à memória em cada instrução e/ou à complexidade do cálculo de endereços. Esta seção examina as técnicas de endereçamento mais comuns:

- Endereçamento imediato
- Endereçamento direto
- Endereçamento indireto
- Endereçamento de registrador
- Endereçamento indireto via registrador
- Endereçamento por deslocamento
- Endereçamento a pilha

Esses modos de endereçamento são mostrados na Figura 10.1. Empregamos a seguinte notação:

A = conteúdo de campo de endereço da instrução
R = conteúdo de campo de endereço que referencia um registrador
EA = endereço real (efetivo) da posição que contém o operando
(X) = conteúdo da posição de endereço *X*

A Tabela 10.1 indica como é feito o cálculo de endereço para cada modo de endereçamento.

Antes de iniciar essa discussão, precisamos fazer dois comentários. O primeiro é que quase todas as arquiteturas de computadores fornecem mais de um modo de endereçamento. Surge então a questão de como a unidade de controle pode determinar o modo de endereçamento a ser usado em uma determinada instrução. Diversas abordagens podem ser adotadas. Freqüentemente, códigos de operação diferentes usam modos de endereçamento diferentes. Além disso, um ou mais bits do formato de instrução podem constituir um *campo de modo de endereçamento.* O valor desse campo determina o modo de endereçamento que deve ser usado.

O segundo comentário diz respeito à interpretação do endereço efetivo (EA). Em um sistema sem memória virtual, o *endereço efetivo* é um endereço de memória principal ou um registrador. Em um sistema de memória virtual, ele é um endereço virtual ou um registrador. O mapeamento para endereço físico é função do mecanismo de paginação e é invisível para o programador.

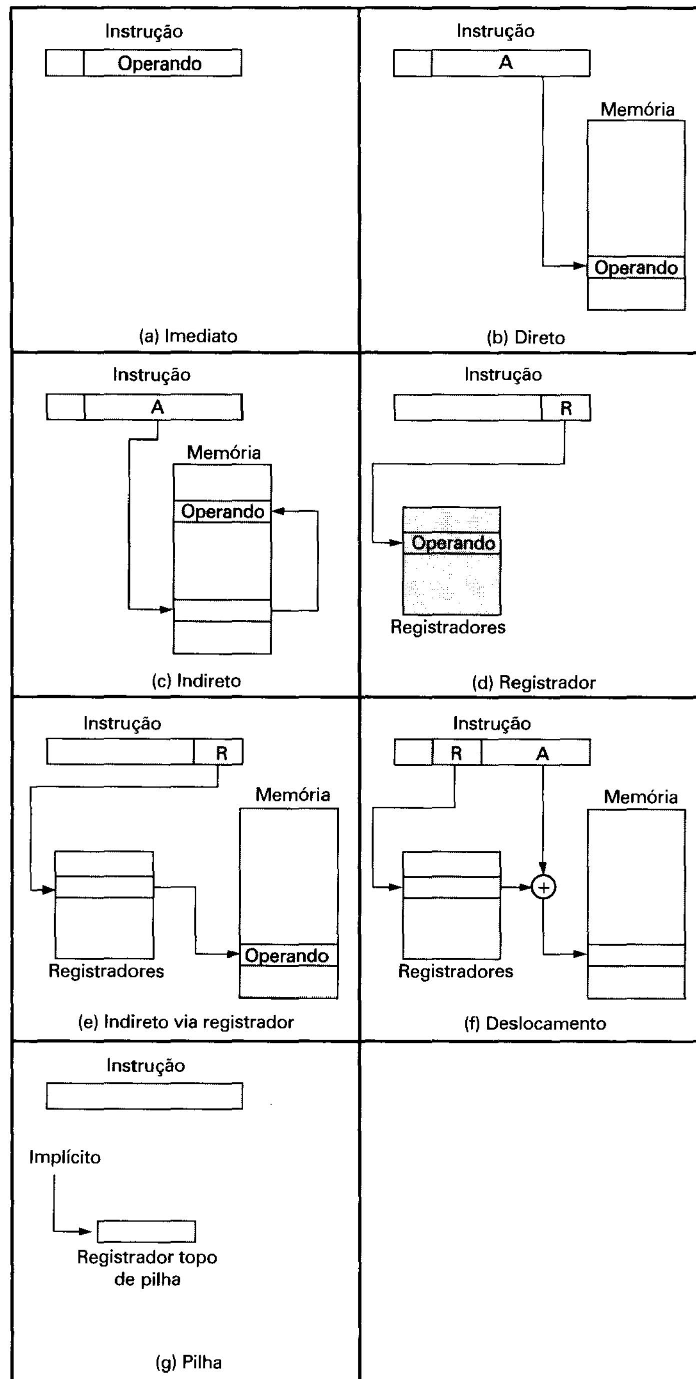


Figura 10.1 Modos de endereçamento.

Tabela 10.1 Modos de endereçamento básicos

Modo	Algoritmo	Principal vantagem	Principal desvantagem
Imediato	Operando = A	Nenhuma referência à memória	Limitada magnitude do operando
Direto	EA = A	Simples	Espaço de endereçamento limitado
Indireto	EA = (A)	Espaço de endereçamento grande	Múltiplas referências à memória
Registrador	EA = R	Nenhuma referência à memória	Espaço de endereçamento limitado
Indireto via registrador	EA = (R)	Espaço de endereçamento grande	Referência extra à memória
Deslocamento	EA = A + (R)	Flexibilidade	Complexidade
Pilha	EA = topo da pilha	Nenhuma referência à memória	Aplicabilidade limitada

Endereçamento imediato

A forma mais simples de endereçamento é o endereçamento imediato, no qual o valor do operando é especificado diretamente na instrução.

$$\text{OPERANDO} = \text{A}$$

Esse modo pode ser usado para definir e usar constantes ou para atribuir valores iniciais em variáveis. O número tipicamente é armazenado na representação em complemento de dois; o bit mais à esquerda do campo de operando é usado como bit de sinal. Quando o operando é carregado em um registrador de dados, o bit de sinal é propagado para a esquerda até preencher todo o registrador.

A vantagem do endereçamento imediato é não requerer qualquer acesso à memória para obter o operando além do acesso para obter a própria instrução, economizando, portanto, um ciclo de cache ou de memória no ciclo de instrução. A desvantagem é que o tamanho do operando é limitado pelo tamanho do campo de endereço, o qual, na maioria dos conjuntos de instruções, é bem menor que o tamanho de uma palavra.

Endereçamento direto

Uma forma muito simples de endereçamento é o endereçamento direto, no qual o campo de endereço contém o endereço efetivo do operando:

$$\text{EA} = \text{A}$$

Esse modo de endereçamento era comum nas primeiras gerações de computadores e ainda é encontrado em diversos computadores de pequeno porte. Ele requer apenas um acesso à memória e nenhum cálculo especial. Sua limitação óbvia é fornecer um espaço de endereçamento limitado.

Endereçamento indireto

No endereçamento direto, o tamanho do campo de endereço normalmente é menor que o tamanho de uma palavra, o que limita a faixa de endereços que podem ser especificados. Uma possível solução para esse problema é especificar, no campo de endereço, o endereço de uma palavra de memória, que, por sua vez, contém o endereço do operando. Esse modo é conhecido como *endereçamento indireto*:

$$EA = (A)$$

Como dissemos anteriormente, a expressão (A) deve ser interpretada como o conteúdo da posição (endereço) A. A vantagem óbvia dessa abordagem é que, se uma palavra tem N bits, o espaço de endereçamento agora disponível tem tamanho 2^N. A desvantagem é que são necessários dois acessos à memória para obter o operando da instrução: o primeiro para obter o endereço do operando e o segundo para obter seu valor.

Embora o número de palavras que podem ser endereçadas seja agora igual a 2^N, o número de endereços efetivos distintos que podem ser referenciados em qualquer instante é limitado a 2^K, onde K é o tamanho do campo de endereço. Essa limitação tipicamente não é muito restritiva e pode ser tolerada. Em um ambiente de memória virtual, todas as posições de endereço efetivo de um processo podem ser confinadas à página 0 de qualquer processo. Como o campo de endereço de uma instrução é pequeno, elas produzirão naturalmente endereços diretos de baixa numeração, que pertencem à página 0. A única restrição é que o tamanho da página deve ser maior ou igual a 2^K. Enquanto o processo estiver ativo, ocorrerão repetidas referências à página 0, fazendo com que essa página permaneça na memória real. Portanto, uma referência indireta à memória envolverá, no máximo, uma falta de página em vez de duas.

Uma variante de endereçamento indireto raramente usada é o endereçamento indireto de múltiplos níveis ou em cascata:

$$EA = (...(A)...)$$

Nesse caso, um bit da palavra de endereço é um indicador de endereçamento indireto (I). Se o bit I for 0, a palavra contém o endereço efetivo EA. Se o bit I for 1, então outro nível de endereçamento indireto é usado. Parece não existir qualquer vantagem particular nessa abordagem e sua desvantagem é que podem ser requeridos três ou mais acessos à memória para obter o operando.

Endereçamento de registrador

O endereçamento de registrador é semelhante ao endereçamento direto. A única diferença é que o campo de endereço se refere a um registrador e não a um endereço na memória principal:

$$EA = R$$

Tipicamente, um campo de endereço que referencia um registrador tem 3 a 4 bits, possibilitando referenciar um total de 8 a 16 registradores de propósito geral.

As vantagens do endereçamento de registrador são: (1) o tamanho do campo de endereço requerido na instrução é pequeno; e (2) não requer nenhuma referência à memória. Como foi discutido no Capítulo 4, o tempo de acesso a um registrador interno da CPU é muito me-

nor que o tempo de acesso à memória principal. A desvantagem do endereçamento de registrador é que o espaço de endereçamento é muito limitado.

Se o modo de endereçamento de registrador for muito usado em um conjunto de instruções, isso significará que os registradores da CPU serão muito usados. Como o número de registradores é rigidamente limitado (em comparação ao número de posições de memória principal), apenas faz sentido utilizá-los dessa maneira se eles forem usados eficientemente. Se, a cada operação, o operando for trazido da memória para um registrador, usado na operação e em seguida armazenado de volta na memória, um passo intermediário dispendioso terá sido adicionado. Em vez disso, se o operando permanecer mais tempo no registrador, sendo usado em diversas operações, obteremos assim uma real economia de tempo. Um exemplo disso são os resultados intermediários em um cálculo. Suponha que o algoritmo de multiplicação de números em complemento de dois deva ser implementado em software. No fluxograma da Figura 8.12, a posição rotulada como A é mencionada várias vezes e, portanto, deve ser implementada em um registrador e não em uma posição da memória principal.

Cabe ao programador decidir que valores devem permanecer nos registradores e quais devem ser armazenados na memória principal. A maioria das CPUs modernas emprega múltiplos registradores de propósito geral, deixando a cargo do programador em linguagem de montagem (ou projetista de compiladores) decidir como esses registradores devem ser utilizados.

Endereçamento indireto via registrador

Assim como o endereçamento de registrador é análogo ao endereçamento direto, o endereçamento indireto via registrador é análogo ao endereçamento indireto. Em ambos os casos, a única diferença é que o campo de endereço se refere a um registrador e não a uma posição de memória. Portanto, no endereçamento indireto via registrador,

$$EA = (R)$$

As vantagens e as limitações do endereçamento indireto via registrador são basicamente as mesmas do endereçamento indireto. Ambos são usados para superar a limitação do espaço de endereçamento, dada pelo tamanho do campo de endereço, fazendo com que o campo de endereço referencie uma palavra de memória que contém um endereço. Entretanto, o endereçamento indireto via registrador requer um acesso à memória a menos que o endereçamento indireto.

Endereçamento por deslocamento

Um modo de endereçamento bastante poderoso combina as capacidades do endereçamento direto e do endereçamento indireto via registrador. Ele é conhecido por uma variedade de nomes, dependendo do contexto de uso, embora o mecanismo básico seja o mesmo. Referimo-nos a ele como *endereçamento por deslocamento.*

$$EA = A + (R)$$

O endereçamento por deslocamento requer que a instrução tenha dois campos de endereço, pelo menos um dos quais é explícito. O valor contido em um dos campos de endereço (valor = A) é usado diretamente. O outro campo de endereço, ou uma referência implícita baseada no código de operação, especifica um registrador cujo conteúdo é adicionado a A, para produzir o endereço efetivo.

Os três usos mais comuns do endereçamento por deslocamento são:

- Endereçamento relativo
- Endereçamento via registrador-base
- Indexação

Endereçamento relativo

No endereçamento relativo, o registrador referenciado implicitamente é o contador de programa (PC). Isto é, o endereço da instrução corrente é adicionado ao campo de endereço para produzir o endereço efetivo EA. Nessa operação, o campo de endereço tipicamente é tratado como um número em complemento de dois. Portanto, o endereço efetivo é um deslocamento relativo ao endereço da instrução.

O endereçamento relativo explora o conceito de localidade discutido nos Capítulos 4 e 7. Se a maioria das referências à memória é para posições relativamente próximas à instrução que está sendo executada, então o uso de endereçamento relativo economiza bits de endereço na instrução.

Endereçamento via registrador-base

No endereçamento via registrador-base, a interpretação é a seguinte: o registrador referenciado contém um endereço de memória e o campo de endereço, um deslocamento em relação a esse endereço (normalmente representado como um número inteiro sem sinal). A referência ao registrador pode ser explícita ou implícita.

Esse tipo de endereçamento também explora a localidade de referências à memória e constitui uma forma conveniente de implementar a segmentação de memória, discutida no Capítulo 7. Algumas implementações empregam um único registrador-base de segmento, que é referenciado implicitamente. Em outras, o programador pode escolher o registrador que conterá o endereço-base do segmento, sendo este referenciado explicitamente na instrução. Nesse último caso, se o tamanho do campo de endereço for K e o número de registradores que podem ser usados for N, a instrução poderá referenciar qualquer uma das N áreas de 2^K palavras.

Indexação

No modo de indexação (também conhecido como modo indexado), a interpretação tipicamente é a seguinte: o campo de endereço contém um endereço de memória principal e o registrador especificado, um deslocamento positivo relativo a esse endereço. Note que isso é o oposto da interpretação dada no caso do endereçamento via registrador-base. É claro que essa diferença não é apenas uma questão de interpretação do usuário. Como o campo de endereço é considerado um endereço de memória no modo de indexação, ele geralmente contém maior número de bits que o campo de endereço de uma instrução com endereçamento via registrador-base. Além disso, veremos que existem alguns refinamentos para a indexação que não seriam úteis no contexto de endereçamento via registrador-base. Entretanto, o método de cálculo do endereço efetivo é o mesmo nos dois casos e em ambos a referência ao registrador algumas vezes é explícita e outras vezes é implícita (para diferentes tipos de CPU).

Um uso importante da indexação é como um mecanismo eficiente para implementar operações iterativas. Considere, por exemplo, uma lista de números armazenados a partir de

um endereço A. Suponha que queremos adicionar 1 a cada elemento da lista. É necessário buscar cada valor na memória, adicionar 1 a esse valor e armazená-lo de volta na memória. A seqüência de endereços efetivos que devem ser referenciados é A, A + 1, A + 2,... , até a última posição na lista. Isso pode ser feito facilmente usando a indexação. O valor A é armazenado no campo de endereço da instrução e o registrador escolhido, denominado *registrador índice*, é inicializado com valor 0. Depois de cada operação, o registrador índice é incrementado de 1.

Como registradores índice são usados comumente para essas tarefas iterativas, é comum haver necessidade de incrementar ou decrementar o valor contido nesse registrador, depois de cada referência a ele. Em razão de essa operação ser comum, ela é feita automaticamente em alguns sistemas, como parte do mesmo ciclo de instrução. Essa técnica é conhecida como *auto-indexação*. Se determinados registradores são exclusivamente reservados para a indexação, a auto-indexação pode ser invocada implícita e automaticamente. Se forem usados registradores de propósito geral, pode ser necessário indicar se deve ou não ser efetuada a auto-indexação, reservando um bit da instrução para esse fim. A auto-indexação com incremento pode ser representada como a seguir:

$$EA = A + (R)$$
$$R \leftarrow (R) + 1$$

Algumas máquinas fornecem tanto endereçamento indireto quanto indexação, sendo possível empregar ambos na mesma instrução. Existem duas possibilidades: a indexação pode ser feita antes ou depois do endereçamento indireto.

Se a indexação for feita depois do endereçamento indireto, a técnica será denominada *pós-indexação*:

$$EA = (A) + (R)$$

Primeiramente, o conteúdo do campo de endereço é usado para acesso à posição de memória que contém o endereço direto. Esse endereço é então indexado pelo valor do registrador. Essa técnica é útil para endereçar um dentre um conjunto de blocos de dados, de um formato fixo. Por exemplo, como mencionamos no Capítulo 7, o sistema operacional tem de manter um bloco de controle de processo para cada processo. As operações efetuadas sobre um bloco são as mesmas, independentemente do bloco que está sendo manipulado. Dessa maneira, em instruções que manipulam esses blocos, o campo de endereço pode apontar para uma posição de memória (valor = A) que contém um apontador para o início do bloco de controle de processo e o registrador índice conteria um deslocamento dentro do bloco.

Na *pré-indexação*, a indexação é feita antes do endereçamento indireto:

$$EA = (A + (R))$$

O endereço é calculado como na indexação simples. Entretanto, nesse caso, o endereço calculado não contém o operando, mas o endereço do operando. Um exemplo do uso dessa técnica é na construção de uma tabela de múltiplos endereços de desvio. Em um determinado ponto de um programa, pode ocorrer o desvio para diferentes instruções, dependendo do resultado de um teste de condição. Isso pode ser implementado por meio de uma tabela de endereços, carregada em uma determinada posição de memória A. Indexando-se essa tabela, o endereço requerido pode ser encontrado.

Tipicamente, um conjunto de instruções não inclui conjuntamente os dois modos de indexação (pré-indexação e pós-indexação).

Endereçamento a pilha

O último modo de endereçamento que consideramos é o endereçamento a pilha. Como foi definido no Apêndice 9A, uma pilha consiste em uma seqüência linear de posições de memória, também conhecida como *lista pushdown* ou *fila LIFO* (*last-in-first-out*). A pilha é um bloco reservado de posições de memória. Itens podem ser colocados no topo da pilha, de modo que, a qualquer instante, esse bloco esteja parcialmente preenchido. É associado à pilha um apontador, que contém o endereço do item no topo da pilha. Alternativamente, os dois elementos no topo da pilha podem ser mantidos em registradores da CPU e, nesse caso, o apontador de topo da pilha indica o terceiro elemento da pilha (Figura 9.14b). O apontador de topo da pilha (conhecido como apontador de pilha ou *stack pointer*) é mantido em um registrador. Portanto, referências a posições de memória na pilha são feitas, de fato, por endereçamento indireto via registrador.

O modo endereçamento a pilha é uma forma de endereçamento implícito. As instruções de máquina não precisam incluir uma referência à memória, operando implicitamente sobre o topo da pilha. O uso de pilhas não era muito comum, mas está se tornando bastante comum em microprocessadores.

10.2 MODOS DE ENDEREÇAMENTO DO PENTIUM II E DO PowerPC

Modos de endereçamento do Pentium II

Lembre-se (veja Figura 7.22) de que o mecanismo de tradução de endereços do Pentium II produz um endereço, denominado endereço virtual ou endereço efetivo, que é um endereço relativo ao início de um segmento. A soma do endereço inicial do segmento com o endereço efetivo produz um endereço linear. Se for usada paginação, esse endereço linear deve passar por meio de um mecanismo de tradução de endereço de página para produzir um endereço físico. Na descrição a seguir, esse último passo será ignorado, uma vez que é transparente para o conjunto de instruções e para o programador.

O Pentium II é equipado com uma variedade de modos de endereçamento, visando possibilitar a execução eficiente de linguagens de alto nível. A Figura 10.2 indica o hardware envolvido. O segmento referenciado é determinado pelo registrador de segmento. Existem seis registradores de segmento. O registrador que é usado em uma referência particular depende do contexto de execução e da instrução. Cada registrador de segmento mantém o endereço inicial do segmento correspondente. Associado a cada registrador de segmento visível ao usuário existe um registrador descritor de segmento (não visível para o programador), que registra direitos de acesso ao segmento, assim como o endereço inicial e final (tamanho) do segmento. Além disso, existem dois registradores que podem ser usados no cálculo de um endereço: o registrador-base e o registrador índice.

A Tabela 10.2 apresenta os modos de endereçamento do Pentium II. Cada um deles é descrito a seguir.

No **modo imediato**, o operando é incluído na instrução. O operando pode ser um byte, uma palavra ou uma palavra dupla.

No **modo de operando registrador**, o operando é localizado em um registrador. Para instruções de caráter geral, tais como transferência de dados e instruções aritméticas ou lógicas, o operando pode ser um dos registradores de propósito geral de 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), 16 bits (AX, BX, CX, DX, SI, SP, BP) ou 8 bits (AH, BH, CH, DH, AL, BL, CL, DL). Para operações de ponto flutuante, os operandos de 64 bits são formados pelo uso de um par de registradores de 32 bits. Existem também algumas instruções que referenciam os registradores de segmento (CS, DS, ES, SS, FS, GS).

Os demais modos de endereçamento referenciam posições de memória. Uma posição de memória deve ser especificada em termos do segmento que contém a posição e do seu endereço relativo ao início do segmento. Em alguns casos, o segmento é especificado explicitamente; em outros, é especificado por regras simples que determinam um segmento padrão.

No **modo de endereçamento por deslocamento**, o endereço relativo do operando (o endereço efetivo, na Figura 10.2) é especificado como parte da instrução, como um deslocamento de 8, 16 ou 32 bits. Com segmentação, todos os endereços em instruções são simplesmente um endereço relativo dentro de um segmento. O modo de endereçamento por deslocamento é usado em poucas máquinas porque, como dissemos anteriormente, ele implica instruções com tamanho muito grande. No caso do Pentium II, o deslocamento pode ter até 32 bits, resultando em uma instrução de 6 bytes. Esse tipo de endereçamento pode ser útil para referenciar variáveis globais.

Os demais modos de endereçamento são indiretos, no sentido de que o campo de endereço da instrução indica ao processador onde obter o endereço do operando. O **modo base** especifica que o endereço efetivo está contido em um dos registradores de 8, 16 ou 32 bits. Isso é equivalente ao que denominamos, anteriormente, de endereçamento indireto via registrador.

No **modo base com deslocamento**, a instrução inclui um deslocamento a ser adicionado a um registrador-base, que pode ser qualquer registrador de propósito geral. Alguns exemplos de uso desse modo de endereçamento são:

- Usado por compiladores para apontar para o início da área de variáveis locais. Por exemplo, o registrador-base pode apontar para o início de um registro de ativação (*stack frame*), que contém as variáveis locais de um procedimento.
- Usado para indexar um vetor, quando o tamanho dos elementos é diferente de 2, 4 ou 8 bytes, não sendo possível, portanto, indexar o vetor por meio de um registrador índice. Nesse caso, o deslocamento aponta para o início do vetor e o registrador-base mantém valores resultantes do cálculo que determina o endereço relativo de cada elemento específico dentro do vetor.
- Usado para endereçar campos de um registro. O registrador-base aponta para o início do registro e o deslocamento é o endereço relativo do campo.

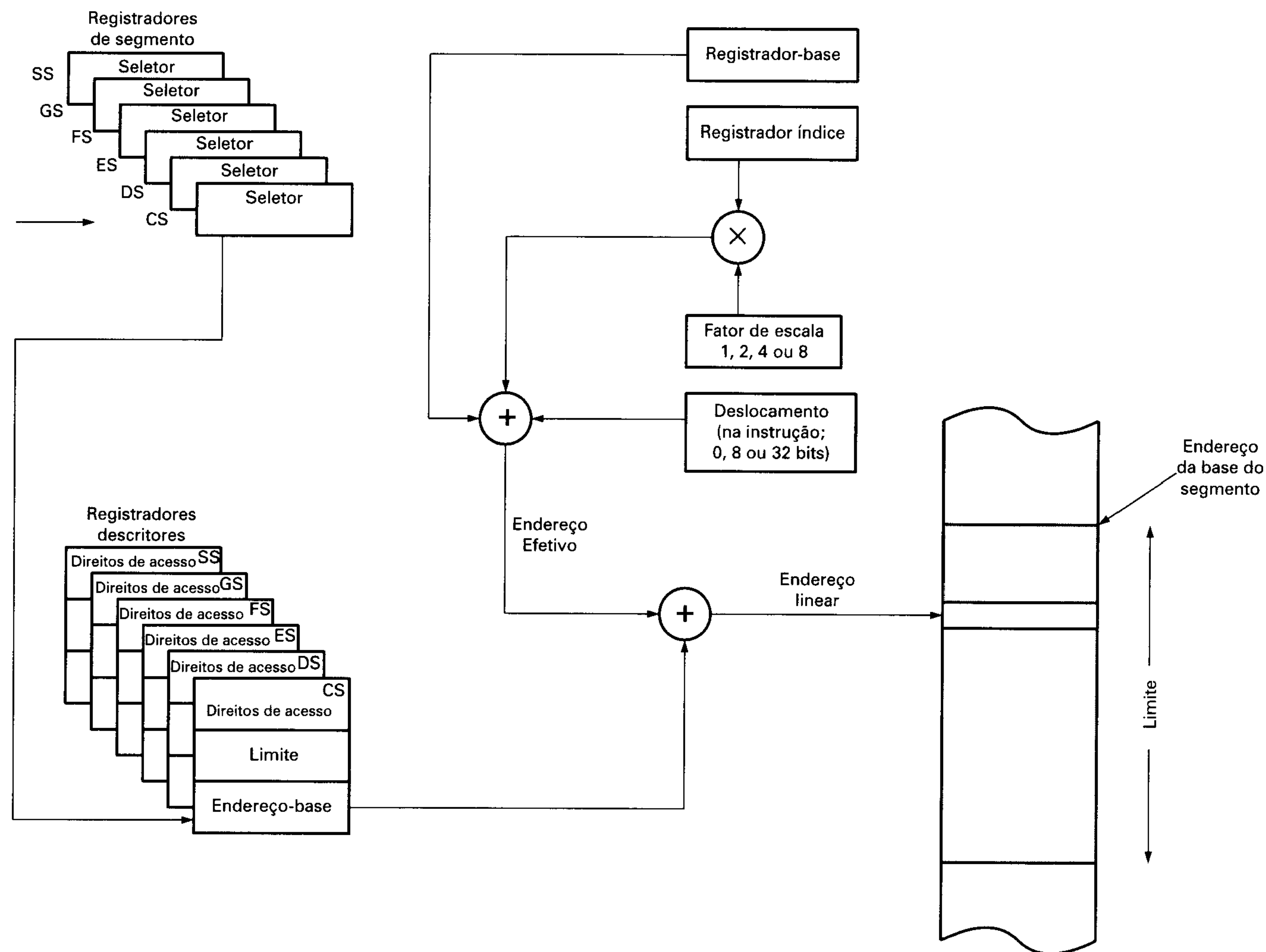


Figura 10.2 Cálculo de modo de endereçamento do Pentium II.

Tabela 10.2 Modos de endereçamento do Pentium II

Modo	Algoritmo
Imediato	Operando = A
Operando registrador	LA = R
Deslocamento	LA = (SR) + A
Base	LA = (SR) + (B)
Base com deslocamento	LA = (SR) + (B) + A
Índice com fator de escala e deslocamento	LA = (SR) + (I) × S + A
Base mais índice e deslocamento	LA = (SR) + (B) + (I) + A
Base mais índice com fator de escala e deslocamento	LA = (SR) + (I) × S + (B) + A
Relativo	LA = (PC) + A

LA = endereço linear
(X) = conteúdo de X
SR = registrador de segmento
PC = contador de programa
A = conteúdo de um campo de endereço na instrução
R = registrador
B = registrador-base
I = registrador índice
S = fator de escala

No **modo índice com fator de escala e deslocamento**, a instrução inclui um deslocamento que é somado a um registrador índice. O registrador índice pode ser qualquer um dos registradores de propósito geral, exceto o registrador ESP, que geralmente é usado para endereçamento a pilha. Para calcular o endereço efetivo, o conteúdo do registrador índice é multiplicado por um fator de escala igual a 1, 2, 4 ou 8 e então é adicionado ao deslocamento. Esse modo de endereçamento é muito conveniente para indexar vetores. O fator de escala 2 pode ser usado para vetores de números inteiros de 16 bits, o fator de escala 4 pode ser usado para vetores de números inteiros de 32 bits ou de números de ponto flutuante e, finalmente, o fator de escala 8 pode ser usado para vetores de números de ponto flutuante de precisão dupla.

No **modo base mais índice e deslocamento**, o endereço efetivo é obtido somando os conteúdos do registrador-base, do registrador índice e do deslocamento. Também nesse caso o registrador-base pode ser qualquer registrador de propósito geral e o registrador índice, qualquer registrador de propósito geral, exceto ESP. Esse modo de endereçamento pode ser usado, por exemplo, para endereçar elementos em um vetor local em um registro de ativação localizado na pilha. Ele também pode ser usado para o acesso a matrizes de duas dimensões; nesse caso, o deslocamento aponta para o início da matriz e cada registrador trata uma dimensão da matriz.

No **modo base mais índice com fator de escala e deslocamento**, o conteúdo do registrador índice é multiplicado por um fator de escala e somado com o conteúdo do registrador-base e o deslocamento. Esse modo de endereçamento é útil se um vetor está armazenado em um registro de ativação; nesse caso, os elementos do vetor devem ter tamanho igual a 2, 4 ou 8 bytes. Ele também fornece indexação eficiente para matrizes de duas dimensões, quando os elementos da matriz têm tamanho igual a 2, 4 ou 8 bytes.

Finalmente, em instruções de transferência de controle pode ser usado **endereçamento relativo**. Um deslocamento é adicionado ao valor do contador de programa, que aponta para a próxima instrução a ser executada. Nesse caso, o deslocamento é tratado como um valor com sinal, com tamanho de um byte, uma palavra ou uma palavra dupla. Portanto, é possível aumentar ou diminuir o valor do endereço no contador de programa.

Modos de endereçamento do PowerPC

Assim como boa parte das máquinas RISC, e diferentemente do Pentium II e da maioria das máquinas CISC, o PowerPC usa um conjunto simples e relativamente direto de modos de endereçamento. Como indica a Tabela 10.3, esses modos são convenientemente classificados conforme o tipo de instrução.

Instruções de carga e armazenamento

O PowerPC oferece dois modos de endereçamento alternativos para instruções de carga e armazenamento (*load/store*) (Figura 10.3). No **endereçamento indireto**, a instrução inclui um deslocamento de 16 bits, a ser adicionado a um registrador-base, que pode ser qualquer um dos registradores de propósito geral. Além disso, a instrução pode especificar que o endereço efetivo recém-computado deve ser armazenado no registrador-base, atualizando seu conteúdo. Essa opção de atualização é útil para a indexação progressiva de vetores em laços de repetição.

Tabela 10.3 Modos de endereçamento do PowerPC

Modo	Algoritmo
	Instruções de carga e armazenamento
Indireto	EA = (BR) + D
Indexado indireto	EA = (BR) + (IR)
	Instruções de desvio
Absoluto	EA = I
Relativo	EA = (PC) + I
Indireto	EA = (L/CR)
	Instruções aritméticas de ponto fixo
Registrador	EA = GPR
Imediato	Operando = I
	Instruções aritméticas de ponto flutuante
Registrador	EA = FPR

EA = endereço efetivo
(X) = conteúdo de X
BR = registrador-base
IR = registrador índice
L/CR = registrador de ligação/registrador contador
GPR = registrador de propósito geral
FPR = registrador de ponto flutuante
D = deslocamento
I = valor imediato
PC = contador de programa

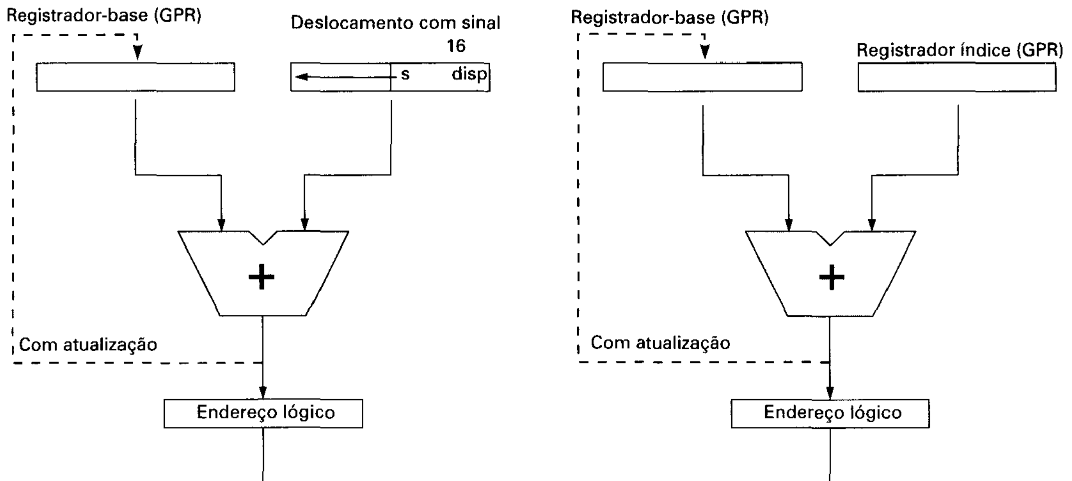


Figura 10.3 Modos de endereçamento a operandos na memória do PowerPC (Diefendorff, 1994b).

A outra técnica de endereçamento para instruções de carga e armazenamento é o **endereçamento indexado indireto**. Nesse caso, a instrução especifica um registrador-base e um registrador índice, ambos podendo ser qualquer um dos registradores de propósito geral. O endereço efetivo é a soma dos conteúdos desses dois registradores. Também nesse caso existe a opção de atualizar o conteúdo do registrador-base com o endereço efetivo calculado.

Instruções de desvio

Três modos de endereçamento são oferecidos para instruções de desvio. Quando é usado o **endereçamento absoluto** em instruções de desvio incondicional, o endereço efetivo da próxima instrução é obtido a partir de um valor imediato de 24 bits contido na instrução. Esse valor de 24 bits é estendido para 32 bits, adicionando dois zeros nas duas posições de bit menos significativas e estendendo o sinal (isso é feito porque toda instrução deve ser armazenada em endereços múltiplos de 32 bits). Para instruções de desvio condicional, o endereço efetivo da próxima instrução é obtido a partir de um valor imediato de 16 bits contido na instrução. Esse valor é também estendido para um valor de 32 bits, adicionando dois zeros nas duas posições de bit menos significativas e estendendo o sinal.

No **endereçamento relativo**, o valor imediato de 24 bits (para instruções de desvio incondicional) ou o valor imediato de 14 bits (para instruções de desvio condicional) é também estendido como antes. O valor resultante é então adicionado ao contador de programa, sendo portanto um endereço relativo ao endereço da instrução corrente. Outro possível modo de endereçamento em instruções de desvio condicional é o **endereçamento indireto**. Nesse modo, o endereço da próxima instrução é obtido no registrador de ligação ou no registrador contador. Note que, nesse caso, o registrador contador é usado para conter o endereço para uma instrução de desvio. Além disso, pode ser usado para manter um contador de interações de um laço de repetição, como explicado anteriormente.

Instruções aritméticas

Nas instruções de aritmética de números inteiros, todos os operandos devem estar contidos em registradores ou ser especificados como parte da instrução. No **endereçamento de registrador**, o operando fonte ou destino pode ser qualquer um dos registradores de propósito geral. No **endereçamento imediato**, o operando fonte é um valor de 16 bits, com sinal contido na instrução.

Nas instruções aritméticas de ponto flutuante, todos os operandos devem estar em registradores de ponto flutuante, isto é, apenas o modo de **endereçamento de registrador** pode ser usado.

10.3 FORMATOS DE INSTRUÇÃO

O formato de uma instrução determina a disposição dos bits da instrução, em termos das suas partes constituintes. Um formato de instrução deve incluir um código de operação e zero ou mais operandos, implícita ou explicitamente. Cada operando explícito é referenciado usando um dos modos de endereçamento descritos na Seção 10.1. O formato deve indicar, implícita ou explicitamente, um modo de endereçamento para cada operando. Na maioria dos conjuntos de instruções, é utilizado mais de um formato de instrução.

O projeto de formatos de instrução é uma arte complexa, e uma variedade impressionante de projetos tem sido implementada. As questões fundamentais do projeto de formatos de instrução são discutidas a seguir e alguns projetos são examinados brevemente, para ilustrar pontos específicos. Em seguida, examinamos mais detalhadamente as soluções adotadas no Pentium II e no PowerPC.

Tamanho de instrução

A questão de projeto mais básica é o tamanho do formato de instrução. Essa decisão afeta, e é afetada, pelo tamanho e pela organização da memória, pela estrutura de barramento e pela complexidade e velocidade da CPU. Ela determina a riqueza e a flexibilidade da máquina, tal como é vista pelo programador em linguagem de montagem.

O problema mais óbvio, nesse caso, é o conflito entre o desejo de fornecer um repertório de instruções poderoso e a necessidade de economizar espaço. Os programadores desejam mais códigos de operação, mais operandos, mais modos de endereçamento e maior espaço de endereçamento. Mais códigos de operação e mais operandos tornam mais fácil a tarefa de programar, permitindo escrever programas mais compactos para executar uma dada tarefa. De maneira semelhante, mais modos de endereçamento dão ao programador maior flexibilidade na implementação de certas funções, tais como manipulação de tabelas e desvios com múltiplos endereços de destino. Por último, é claro que, com o aumento do tamanho da memória principal e o uso crescente de memória virtual, os programadores desejam poder endereçar uma área de memória maior. Tudo isso (códigos de operação, operandos, modos de endereçamento, endereços) requer bits na instrução, implicando instruções de tamanho maior. Entretanto, instruções maiores podem significar desperdício. Uma instrução de 32 bits ocupa duas vezes o espaço de uma instrução de 16 bits, sendo, provavelmente, muito menos que duas vezes mais útil que esta última.

Além dessas questões básicas, existem outros aspectos a serem considerados. O tamanho da instrução deve ser igual ao tamanho da unidade de transferência de dados de e para

a memória (em um sistema com barramento, a largura do barramento de dados) ou um deve ser múltiplo do outro. Caso contrário, não é obtido um número inteiro de instruções durante um ciclo de busca. Um aspecto relacionado é a taxa de transferência da memória. Essa taxa não tem acompanhado o aumento de velocidade dos processadores. Assim, a memória pode tornar-se um gargalo, caso o processador execute instruções mais rápido do que é capaz de buscá-las. Uma solução para esse problema é o uso de memória cache (veja a Seção 4.3), outra é usar instruções menores. Instruções de 16 bits podem ser buscadas a uma taxa duas vezes maior que instruções de 32 bits, mas, provavelmente, podem ser executadas menos de duas vezes mais rápido.

Uma característica aparentemente mundana, mas muito importante, é que o tamanho da instrução deve ser múltiplo não só do tamanho de um caractere, que normalmente é de 8 bits, como também do tamanho de um número de ponto fixo. Para entender isso, precisamos usar aquela palavra infelizmente maldefinida, a *palavra* (de memória) (Frailey, 1983). O tamanho de uma palavra de memória é, de certo modo, a unidade 'natural' de organização. Ele normalmente determina o tamanho de números de ponto fixo (em geral os dois são iguais) e, tipicamente, é igual ou é uma fração inteira do tamanho da unidade de transferência de dados da memória. Como o caractere é uma forma de dado comum, é desejável que um número inteiro de caracteres seja armazenado em uma palavra. Caso contrário, quando fossem armazenados múltiplos caracteres, alguns bits seriam desperdiçados em cada palavra ou os caracteres não poderiam ser confinados nos limites de uma palavra. A importância desse ponto é tal que, quando a IBM introduziu o Sistema 360 e quis empregar caracteres de 8 bits, ela tomou a decisão de mudar da arquitetura de 36 bits, usada nos computadores científicos das séries 700/7000, para uma arquitetura de 32 bits.

Alocação de bits

Analisamos anteriormente alguns fatores que influem na decisão do tamanho do formato de instruções. Um problema igualmente difícil é como alocar, nesse formato, os bits de cada campo da instrução. As questões envolvidas são bastante complexas.

Claramente, fixado um tamanho de instrução, existe um conflito entre o número de códigos de operação e a capacidade de endereçamento. Um maior número de códigos de operação significa, obviamente, mais bits no campo de código de operação. Isso reduz o número de bits disponíveis para endereçamento. Um refinamento interessante desse problema é o uso de códigos de operação de tamanho variável. Nessa abordagem, existe um tamanho mínimo para o código de operação, mas, em alguns casos, podem ser especificadas operações adicionais, usando bits adicionais na instrução. Para instruções de tamanho fixo, isso implica menor número de bits para endereçamento. Portanto, essa característica é usada apenas para instruções que requerem poucos operandos e/ou um modo de endereçamento menos poderoso.

Os seguintes fatores, todos relacionados, determinam o uso dos bits de endereçamento:

- **Número de modos de endereçamento:** algumas vezes, o modo de endereçamento pode ser indicado implicitamente. Por exemplo, certos códigos de operação podem especificar sempre o uso de indexação. Em outros casos, o modo de endereçamento deve ser explícito, requerendo um ou mais bits para especificá-lo.
- **Números de operandos:** vimos anteriormente que o uso de instruções com menor número de operandos pode resultar em programas maiores e mais complicados (veja, por exemplo, a Figura 9.3). Instruções típicas de máquinas atuais têm dois ope-

randos. O endereço de cada operando pode requerer seu próprio indicador de modo de endereçamento ou pode ser usado um único indicador de modo de endereçamento relativo a apenas um dos campos de endereço.

- **Memória ou registrador:** toda máquina deve ter registradores, para possibilitar que sejam trazidos dados para a CPU para processamento. Se existir apenas um registrador visível para o usuário (normalmente chamado de acumulador), o endereço de um operando será implícito, não consumindo bits da instrução. Entretanto, a programação com um único registrador é ineficaz e requer muitas instruções. Mesmo usando múltiplos registradores, é necessário apenas um pequeno número de bits para especificar o registrador. Quanto maior for o número de operandos em registradores, tanto menor será o número de bits de endereço necessários. Diversos estudos indicam que é desejável um total de 8 a 32 registradores visíveis para o usuário (Lunde, 1977; Huck, 1983).
- **Número de conjuntos de registradores:** diversas máquinas possuem um conjunto de registradores de propósito geral, tipicamente com 8 ou 16 registradores. Esses registradores podem ser usados para armazenar dados ou endereços (para endereçamento por deslocamento). A tendência atual não é utilizar um único banco de registradores de propósito geral, mas fornecer dois ou mais conjuntos de registradores especializados (tais como registradores de dados e registradores de deslocamento). Essa tendência abrange desde microprocessadores até supercomputadores. Uma vantagem dessa abordagem é que, para um número fixo de registradores, essa divisão funcional requer menor número de bits na instrução para especificar um registrador. Por exemplo, com dois conjuntos de oito registradores, são requeridos apenas 3 bits para identificar um registrador; o código de operação determina, implicitamente, o conjunto de registradores referenciado. Parece não haver nenhuma desvantagem nessa abordagem (Lunde, 1977). Em sistemas como o 370 da IBM, que fornece um único conjunto de registradores de propósito geral, os programadores normalmente estabelecem convenções que fixam metade dos registradores para dados e metade para deslocamentos (Mallach, 1979).
- **Faixa de endereços:** para endereços que referenciam a memória, a faixa de endereços que podem ser especificados é relacionada ao número de bits do campo de endereço. Como isso impõe uma limitação severa, o endereçamento direto raramente é usado. No endereçamento por deslocamento, a área de memória endereçável depende do tamanho do registrador de endereço. Além disso, é ainda conveniente permitir o uso de um deslocamento grande em relação ao endereço contido no registrador, o que requer um número relativamente grande de bits de endereço na instrução.
- **Granularidade de endereçamento:** para referências à memória, outro aspecto importante é a granularidade de endereçamento. Em um sistema com palavras de 16 ou 32 bits, um endereço pode referenciar uma palavra ou um byte à escolha do projetista. O endereçamento de byte é conveniente para a manipulação de caracteres, mas requer, para uma memória de tamanho fixo, maior número de bits de endereço.

O projetista se vê, portanto, diante de um conjunto de fatores a considerar e equilibrar. Ainda não está muito claro até que ponto são críticas as várias escolhas. Como exemplo disso, citamos o estudo de Cragon (1979), que compara várias abordagens de formato de instrução, incluindo o uso de uma pilha, registradores de propósito geral, um acumulador e abordagens

que forçam o uso de operandos apenas em registradores. Utilizando um conjunto coerente de hipóteses, não foi observada nenhuma diferença significativa no tamanho do código ou no tempo de execução, para as diferentes abordagens.

Examinamos brevemente como esses vários fatores são equilibrados, no projeto de duas máquinas diferentes.

PDP-8

Um dos projetos de formato de instrução mais simples para um computador de propósito geral foi o do PDP-8 (Bell, 1978b). O PDP-8 usa instruções de 12 bits e opera sobre palavras de 12 bits. Existe um único registrador de propósito geral, o acumulador.

Apesar das limitações desse projeto, o endereçamento é bastante flexível. Cada referência à memória consiste em 7 bits, além de dois modificadores de 1 bit. A memória é dividida em páginas de tamanho fixo, cada uma com $2^7 = 128$ palavras. O cálculo de endereços é baseado em referências para a página 0 ou para a página corrente (a página que contém a instrução), conforme determinado pelo bit de página. O segundo bit modificador indica se deve ser usado endereçamento direto ou indireto. Esses dois modos podem ser combinados, de modo que um endereço indireto seja um endereço de 12 bits, contido em uma palavra da página 0 ou da página corrente. Além disso, oito palavras dedicadas da página 0 são 'registradores' de auto-indexação. Quando é feita uma referência indireta para uma dessas posições, ocorre a pré-indexação.

A Figura 10.4 mostra os formatos de instrução do PDP-8. Existe um código de operação de 3 bits e três tipos de instrução. Para os códigos de operação de 0 a 5, o formato da instrução inclui uma única referência à memória, com um bit de página e um bit de endereçamento indireto. Existem, portanto, apenas seis operações básicas. Para ampliar o grupo de operações, o código de operação 7 especifica uma referência a registrador ou *microinstrução*. Nesse formato, os bits restantes são usados para codificar operações adicionais. Em geral, cada bit define uma operação específica (por exemplo, atribuir valor zero ao acumulador) e esses bits podem ser combinados em uma única instrução. A estratégia da microinstrução foi usada pela DEC desde o PDP-1 e é, em certo sentido, precursora das máquinas microprogramadas de hoje, que serão discutidas na Parte 4. O código de operação 6 indica uma operação de E/S; 6 bits são usados para selecionar um dos 64 dispositivos e 3 bits especificam um comando de E/S particular.

O formato de instruções do PDP-8 é notavelmente eficiente. Ele dá suporte para endereçamento indireto, endereçamento por deslocamento e indexação. Com o uso de extensão do código de operação, é possível especificar um total de aproximadamente 35 instruções. Dada a restrição de 12 bits para o tamanho de uma instrução, os projetistas dificilmente poderiam ter feito melhor.

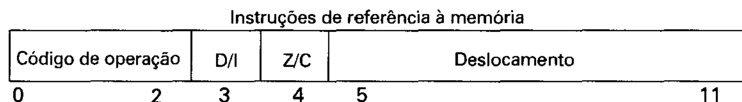
PDP-10

Em marcante contraste com o conjunto de instruções do PDP-8, está o conjunto de instruções do PDP-10. O PDP-10 foi projetado para ser um sistema de tempo compartilhado de grande escala, com ênfase em tornar o sistema fácil de programar, mesmo que isso envolvesse custos adicionais de hardware.

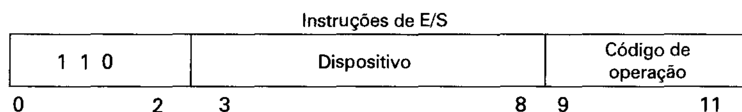
Entre os princípios empregados no projeto do conjunto de instruções estavam os seguintes (Bell, 1978c):

- **Ortogonalidade:** o princípio de ortogonalidade determina que quaisquer duas variáveis de um problema devem ser independentes uma da outra. No contexto de conjunto de instruções, esse termo indica que os demais elementos de uma instrução devem ser independentes do código de operação (não determinados por ele). Os projetistas do PDP-10 usavam esse termo para referir-se ao fato de que um endereço sempre é calculado do mesmo modo, independentemente do código de operação. Essa abordagem contrasta com a adotada em muitas máquinas, em que o modo de endereçamento muitas vezes depende implicitamente do operador que está sendo usado.

Instruções de referência à memória

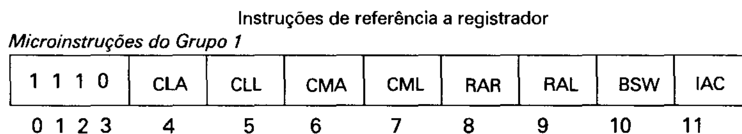


Instruções de E/S

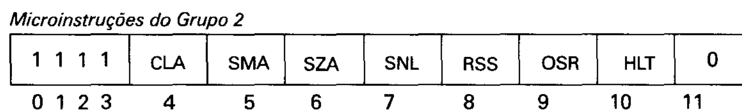


Instruções de referência a registrador

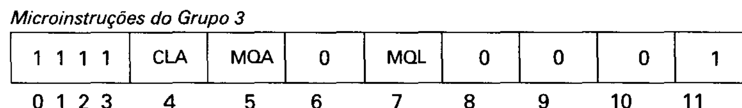
Microinstruções do Grupo 1



Microinstruções do Grupo 2



Microinstruções do Grupo 3



Mnemônicos

CLA = atribui valor zero ao acumulador
CLL = atribui valor zero ao registrador de ligação
CMA = complementa o acumulador
CML = complementa o registrador de ligação
RAR = rotação à direita do acumulador
RAL = rotação à esquerda do acumulador
BSW = troca de byte
IAC = incrementa o acumulador
SMA = salta se o acumulador é negativo
SZA = salta se o acumulador é igual a zero
SNL = salta se o registrador de ligação é diferente de zero
RSS = inverte o sentido do salto
OSR = operação lógica ou com o registrador de troca
HLT = parar
MQA = multiplicador e quociente no acumulador
MQL = carrega multiplicador e quociente

Figura 10.4 Formatos de instrução do PDP-8.

- **Completeza:** cada tipo de dado aritmético (número inteiro, de ponto fixo ou ponto flutuante) deveria ter um conjunto completo e idêntico de operações.
- **Endereçamento direto:** o modo de endereçamento-base com deslocamento, que deixa a cargo do programador a tarefa de organização da memória, foi evitado em favor do endereçamento direto.

Cada um desses princípios tem como principal objetivo facilitar a tarefa de programação.

No PDP-10, uma palavra e uma instrução têm, ambos, tamanho de 36 bits. O formato fixo de instruções é mostrado na Figura 10.5. O código de operação ocupa 9 bits, possibilitando especificar até 512 operações. De fato, são definidas apenas 365 instruções distintas. A maioria das instruções tem dois endereços, um dos quais é um dos 16 registradores de propósito geral. Essa referência a operando ocupa, portanto, 4 bits. A outra referência a operando começa com um campo de endereço de memória de 18 bits, que pode ser usado como operando imediato ou endereço de memória. Nesse último caso, é possível usar tanto indexação quanto endereçamento indireto. Os registradores de propósito geral são também usados como registradores índice.

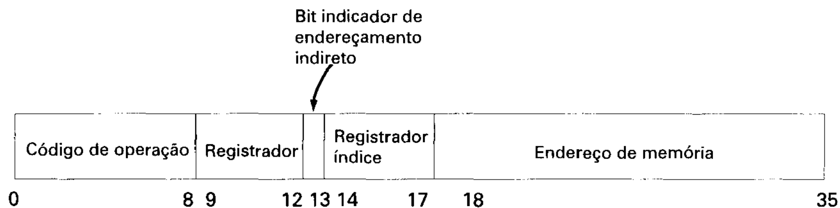


Figura 10.5 Formato de instrução do PDP-10.

Uma instrução com tamanho de 36 bits é uma verdadeira luxúria. Um código de operação de 9 bits é mais que suficiente; não é preciso nenhum esforço para definir mais códigos de operação. O endereçamento também é fácil. Um campo de endereço de 18 bits torna conveniente o endereçamento direto. Para possibilitar o endereçamento de memórias com tamanho maior que 2^{18}, é fornecido endereçamento indireto. Para facilitar a programação, também é fornecido indexação, para a manipulação de tabelas e a implementação de laços de repetição. Quando se dispõe de um campo de operando com 18 bits, o endereçamento imediato torna-se atraente.

O projeto do conjunto de instruções do PDP-10 cumpre os objetivos apresentados anteriormente (Lunde, 1977). O conjunto de instruções do PDP-10 facilita a tarefa do programador e do compilador, ao custo de uma utilização ineficiente do espaço de memória. Essa foi uma escolha consciente dos projetistas e, portanto, não pode ser considerada uma falha de projeto.

Instruções de tamanho variável

Nos exemplos abordados até agora, todas as instruções têm um mesmo tamanho fixo e discutimos, implicitamente, as questões de projeto do formato de instrução nesse contexto. Alternativamente, o projetista pode decidir oferecer uma variedade de instruções com tamanhos diferentes, em vez de instruções com um tamanho único. Essa tática torna fácil fornecer um grande repertório de códigos de operação, com tamanhos diferentes. O endereçamento pode também ser mais flexível, com várias combinações de referência à memória e de regis-

trador juntamente com modos de endereçamento. Usando instruções de tamanho variável, é possível fornecer muitas variações, de maneira compacta e eficiente.

O principal custo dessa abordagem é um aumento da complexidade da CPU. Vários fatores têm contribuído para tornar esse custo menor: diminuição de preço do hardware, uso de microprogramação (discutido na Parte 4) e melhor entendimento dos princípios de projeto da CPU.

Mesmo no caso de instruções de tamanho variável, ainda é conveniente que todos os tamanhos de instrução sejam múltiplos ou divisores inteiros do tamanho da palavra. Como a CPU não conhece o tamanho da próxima instrução a ser buscada, uma estratégia típica é buscar sempre ao menos o número de bytes, ou palavras, correspondente ao tamanho da maior instrução possível. Isso significa que, algumas vezes, são buscadas múltiplas instruções. Como veremos no Capítulo 11, essa é uma boa estratégia em qualquer caso.

PDP-11

O PDP-11 foi projetado para fornecer um conjunto de instruções poderoso e flexível, dentro dos limites de um minicomputador de 16 bits (Bell, 1970).

O PDP-11 possui um conjunto de oito registradores de propósito geral de 16 bits. Dois desses registradores têm significado especial: um deles é usado como apontador de topo da pilha, em operações na pilha, e outro é usado como contador de programa, que mantém o endereço da próxima instrução a ser executada.

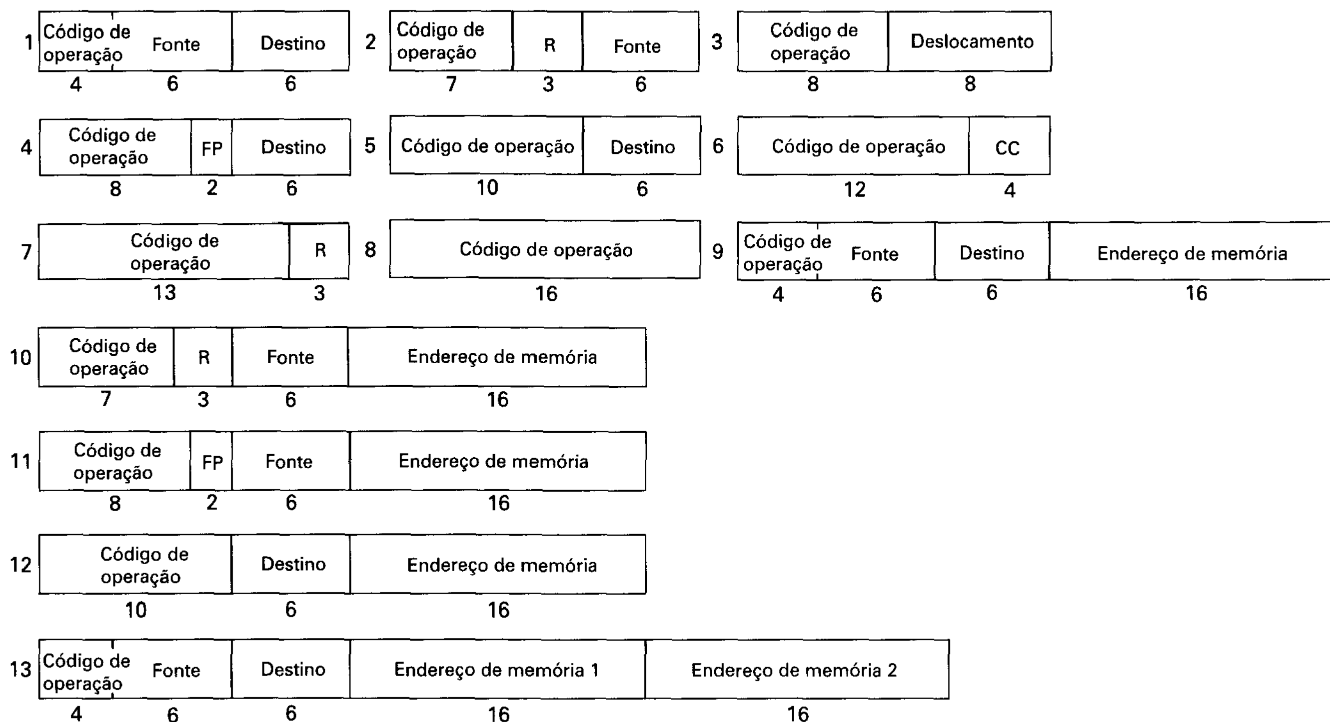
A Figura 10.6 mostra os formatos de instruções do PDP-11. Treze formatos diferentes são usados, incluindo instruções com zero, um e dois endereços. O tamanho do código de operação pode variar de 4 a 16 bits. Uma referência ao registrador tem 6 bits. Três desses bits identificam o registrador e os três restantes identificam o modo de endereçamento. O PDP-11 é dotado de um rico conjunto de modos de endereçamento. Uma vantagem de associar o modo de endereçamento ao operando, e não ao código de operação, é que qualquer modo de endereçamento pode ser usado em qualquer operação. Essa independência, como dissemos anteriormente, é denominada *ortogonalidade*.

As instruções do PDP-11 normalmente têm tamanho de uma palavra (16 bits). Em algumas instruções, são anexados um ou dois endereços de memória, o que inclui instruções de 32 bits e de 48 bits como parte do repertório, para dar maior flexibilidade de endereçamento.

O conjunto de instruções e a capacidade de endereçamento do PDP-11 são bastante complexos. Isso aumenta o custo do hardware e a complexidade de programação. A vantagem é que podem ser desenvolvidos programas mais compactos ou eficientes.

VAX

A maioria das arquiteturas fornece um número relativamente pequeno de formatos de instrução de tamanho fixo. Isso pode causar dois problemas para o programador. O primeiro é que o modo de endereçamento e o código de operação não são ortogonais. Por exemplo, para uma determinada operação, um dos operandos deve estar em um registrador e o outro na memória ou ambos em registradores, e assim por diante. O segundo é que apenas um número limitado de operandos pode ser acomodado na instrução: tipicamente, até dois ou três operandos. Como algumas operações requerem, inerentemente, maior número de operandos, várias estratégias têm de ser adotadas, usando duas ou mais instruções.



Fonte e destino contêm, cada qual, um campo de modo de endereçamento de 3 bits e um número de registrador de 3 bits;
FP é um dos registradores de ponto flutuante, numerados por 0, 1, 2 ou 3;
R é um dos registradores de propósito geral;
CC é um campo de código de condição.

Figura 10.6 Formatos de instrução do PDP-11 (os números indicam o tamanho dos campos).

Para evitar esses problemas, dois critérios foram usados no projeto do formato de instruções do VAX (Strecker, 1978):

1. Todas as instruções devem ter o número 'natural' de operandos.
2. Todos os operandos devem ter a mesma generalidade de especificação.

O resultado é um formato de instrução altamente variável. Uma instrução consiste em um código de operação de 1 ou 2 bytes seguido de zero a seis especificadores de operandos, dependendo do código de operação. O menor tamanho de instrução é 1 byte e podem ser construídas instruções com até 37 bytes. A Figura 10.7 mostra alguns exemplos.

Uma instrução do VAX começa com um código de operação de 1 byte. Esse tamanho é suficiente para a maioria das instruções do VAX. Entretanto, como existem mais de 300 instruções diferentes, 8 bits não são suficientes. Os códigos hexadecimais FD e FF indicam um código de operação estendido, sendo o verdadeiro código de operação especificado no segundo byte.

O restante da instrução consiste em até seis especificadores de operandos. A especificação de um operando tem tamanho de, no mínimo, 1 byte, no qual os 4 bits mais à esquerda especificam o modo de endereçamento. A única exceção para essa regra é o modo literal, indicado pelo padrão 00 nos 2 bits mais à esquerda, e deixando um espaço para um literal de 6 bits. Em virtude dessa exceção, o total de modos de endereçamento distintos que podem ser especificados é igual a 12.

Freqüentemente, um especificador de operando consiste em apenas um byte, com os 4 bits mais à direita especificando um dos 16 registradores de propósito geral. O tamanho do especificador de operando pode ser estendido de duas maneiras. Na primeira, o primeiro byte do especificador de operando pode ser seguido de um valor constante, de um ou mais bytes. Um exemplo é o modo de endereçamento por deslocamento, no qual é usado um deslocamento de 8, 16 ou 32 bits. Na segunda, pode ser utilizado um modo de indexação. Nesse caso, o primeiro byte do especificador de operando é formado do código de modo de endereçamento 0100, de 4 bits, e de um identificador de registrador índice, com 4 bits. O restante do especificador de operando consiste em um especificador de endereço base, que pode ter tamanho de um ou mais bytes.

Você deve estar imaginando que tipo de instrução pode requerer seis operandos. Surpreendentemente, o VAX possui várias instruções desse tipo. Considere, por exemplo, a instrução

```
ADDP6 OP1, OP2, OP3, OP4, OP5, OP6
```

Essa instrução soma dois números na representação decimal empacotada. OP1 e OP2 especificam o tamanho e o endereço inicial de uma das seqüências de dígitos decimais; OP3 e OP4, por sua vez, especificam a segunda seqüência de dígitos. Esses dois números são somados e o resultado é armazenado como uma seqüência de dígitos decimais, cujo tamanho e endereço inicial são especificados por OP5 e OP6.

O conjunto de instruções do VAX tem uma ampla variedade de operações e modos de endereçamento. Isso fornece ao programador, assim como ao projetista de compiladores, uma ferramenta poderosa e flexível para desenvolver programas. Em tese, isso deveria favorecer a compilação de linguagens de alto nível para linguagem de máquina, produzindo código mais eficiente, e resultar, de modo geral, em uso efetivo e eficiente dos recursos da CPU. O preço a ser pago por esses benefícios é um aumento na complexidade da CPU, se comparada a uma CPU com formato e conjunto de instruções mais simples.

Formato hexadecimal	Explicação	Notação em linguagem de montagem e descrição
0 5 (8 bits)	Código de operação para RSB	RSB Retorno de sub-rotina
D 4	Código de operação para CLRL	CLRL R9
5 9	Registrador R9	Zerar o registrador R9
B 0	Código de operação para MOVW	MOVW 356(R4), 25(R11)
C 4	Modo de deslocamento de palavra, Registrador R4	Move a palavra do endereço igual a 356 somado ao conteúdo de R4 para o endereço igual a 25 somado ao conteúdo de R11
6 4 0 1	356 em hexadecimal	
A B	Modo de deslocamento de byte, Registrador R11	
1 9	25 em hexadecimal	
C 1	Código de operação para ADDL3	ADDL3 #5, R0, @ A[R2]
0 5	Literal curto 5	Soma 5 ao número inteiro de 32 bits contido em R0 e armazena o resultado na posição cujo endereço é a soma de A com o valor obtido multiplicando o conteúdo de R2 por 4
5 0	Modo de registrador R0	
4 2	Registrador índice R2	
D F	Endereçamento indireto relativo de palavra (deslocamento em relação ao PC)	
	Deslocamento do PC relativo à posição A	

Figura 10.7 Exemplo de instruções do VAX.

Retornaremos a essas questões no Capítulo 12, no qual examinamos máquinas com conjunto de instruções muito simples.

10.4 FORMATOS DE INSTRUÇÃO DO PENTIUM II E DO PowerPC

Formatos de instrução do Pentium II

O Pentium II possui uma variedade de formatos de instrução. Apenas o campo de código de operação está sempre presente em todos os formatos. A Figura 10.8 mostra o formato geral de instrução. As instruções são constituídas de zero a quatro prefixos de instrução opcionais, um código de operação de um ou dois bytes, um campo de endereço opcional, que consiste no byte Mod r/m e no byte Scale Index, um deslocamento opcional e um campo opcional de operando imediato.

Consideramos, primeiramente, os bytes de prefixo:

- **Prefixos de instrução:** o prefixo de instrução, caso esteja presente, pode ser o prefixo LOCK ou um dos prefixos de repetição. O prefixo LOCK é usado para assegurar o uso exclusivo de memória compartilhada em ambientes de multiprocessamento. Os prefixos de repetição especificam a repetição de uma operação sobre uma seqüência de dados, o que habilita o Pentium II a processar essas seqüências de maneira muito mais rápida do que com um laço de repetição usual, implementado em um programa. Existem cinco prefixos de repetição diferentes: REP, REPE, REPZ, REPNE e REPNZ. Quando o prefixo absoluto REP está presente, a operação especificada na instrução é executada repetidamente sobre elementos sucessivos da seqüência; o número de repetições é especificado pelo registrador CX. O prefixo condicional REP faz com que a instrução seja repetida até que o valor do registrador CX seja igual a zero ou até que uma condição seja satisfeita.
- **Seleção de registrador de segmento:** especifica explicitamente o registrador de segmento que deve ser usado na execução da instrução, em vez do registrador de segmento padrão associado àquela instrução no Pentium II.
- **Tamanho de endereço:** o processador pode endereçar a memória usando endereços de 16 ou 32 bits. O tamanho do campo de endereço determina o tamanho do deslocamento usado nas instruções e o tamanho de endereços relativos usados no cálculo de endereço efetivo. Um tamanho padrão pode ser especificado como 16 ou 32 bits. O prefixo de tamanho de endereço é utilizado para indicar que deve ser usado o tamanho diferente daquele especificado como padrão.
- **Tamanho de operando:** o tamanho padrão de operando pode ser de 16 ou 32 bits; o prefixo de tamanho de operando é utilizado para indicar que deve ser usado o tamanho diferente daquele estabelecido como padrão.

A própria instrução inclui os seguintes campos:

- **Código de operação:** código de um ou dois bytes. O código de operação pode também incluir bits para especificar as seguintes informações: se o dado é um byte ou tem tamanho normal (16 ou 32 bits, dependendo do contexto), a direção de uma operação de transferência de dados (de ou para a memória) e se o valor de um campo de operando imediato deve ser usado com extensão de sinal.

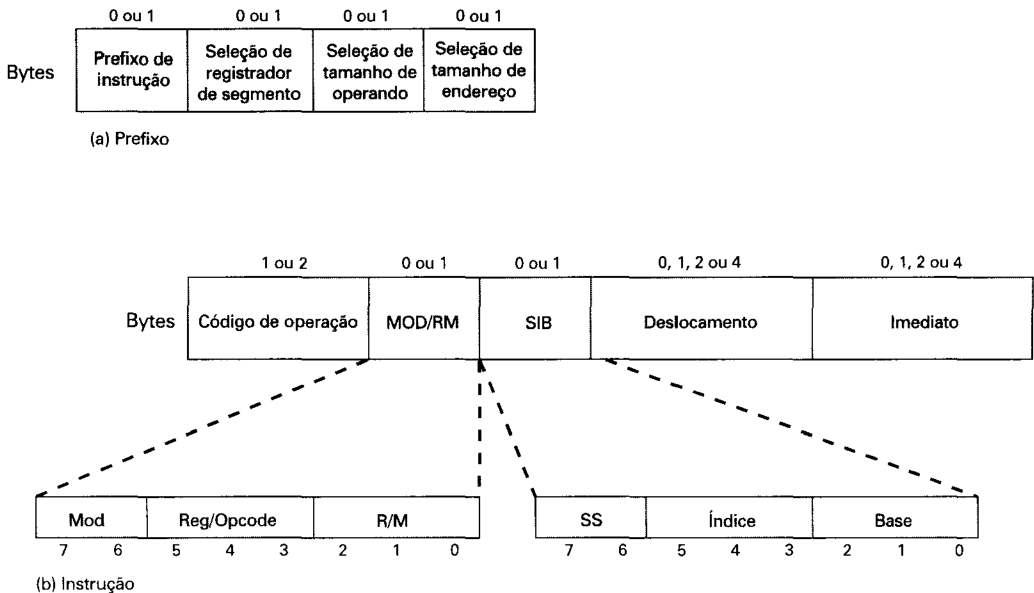


Figura 10.8 Formato de instrução do Pentium II.

- **Mod r/m:** esse e o próximo byte fornecem informação de endereçamento. O byte Mod r/m especifica se o operando está em um registrador ou na memória; se estiver na memória, campos dentro desse byte especificarão o modo de endereçamento a ser usado. Ele consiste em três campos: o campo Mod (2 bits) é combinado com o campo r/m, para formar 32 valores possíveis: 8 registradores e 24 modos de indexação; o campo Reg/Opcode (3 bits) especifica um número de registrador ou três bits adicionais de código de operação; o campo r/m (3 bits) pode especificar um registrador, como a localização de um operando, ou fazer parte da codificação de um modo de endereçamento, em combinação com o campo Mod.
- **SIB:** certas codificações do byte Mod r/m indicam que o modo de endereçamento só é completamente especificado considerando-se também o byte SIB. O byte SIB consiste em três campos: o campo SS (2 bits) especifica um fator de escala para indexação, o campo de índice (3 bits) especifica um registrador índice e o campo de base (3 bits) especifica um registrador-base.
- **Deslocamento:** quando o modo de endereçamento especificado é o modo por deslocamento, é incluído na instrução um campo de deslocamento, que contém um número inteiro com sinal de 8, 16 ou 32 bits.
- **Imediato:** fornece o valor de um operando de 8, 16 ou 32 bits.

Diversas comparações podem ser úteis nesse ponto. No formato do Pentium II, o modo de endereçamento é fornecido como parte do código de operação, e não em cada operando. Como a informação de modo de endereçamento se refere apenas a um operando, pode haver apenas uma referência a operando na memória na instrução. No VAX, ao contrário, o modo de endereçamento é fornecido com cada operando, permitindo operações de memória para memória. As instruções do Pentium II são, portanto, mais compactas. No entanto, se for requerida uma operação de memória para memória, ela pode ser feita no VAX usando uma única instrução.

O formato do Pentium II permite usar na indexação deslocamentos não só de 1 byte, mas também de 2 e 4 bytes. Embora o uso de índices maiores resulte em instruções mais longas, essa característica possibilita maior flexibilidade. Por exemplo, ela é útil no endereçamento de vetores ou registros de ativação muito grandes. O formato de instrução do Sistema/370 da IBM, ao contrário, não permite deslocamentos maiores que 4K bytes (12 bits de informação de deslocamento) e o deslocamento deve ser positivo. Quando uma posição de memória não está dentro do alcance desse deslocamento, o compilador tem de gerar código extra para produzir o endereço necessário. Esse problema é especialmente aparente ao lidar com registros de ativação de procedimentos cujas variáveis locais ocupam espaço maior que 4K bytes. De acordo com Dewar e Smosna (1990), "em virtude dessa restrição, gerar código para o 370 é tão complicado que alguns compiladores para o 370 simplesmente limitam o tamanho do registro de ativação a 4K bytes".

Como se pode observar, a codificação do conjunto de instruções do Pentium II é muito complexa. Isso se deve parcialmente à necessidade de compatibilidade com a máquina 8086 e parcialmente ao desejo dos projetistas de fornecer ao desenvolvedor de compiladores todo o auxílio possível para produzir um código eficiente. É ainda uma questão em aberto se um conjunto de instruções tão complexo como esse é preferível ao seu extremo oposto, representado por um conjunto de instruções RISC.

Formatos de instrução do PowerPC

Todas as instruções do PowerPC têm tamanho de 32 bits e seguem um formato regular. Os primeiros seis bits da instrução especificam a operação a ser efetuada. Em alguns casos, existe uma extensão do código de operação em alguma outra parte na instrução, que especifica um caso particular de uma operação. Na Figura 10.9, os bits do código de operação são representados pela parte sombreada de cada formato.

Observe a estrutura regular dos formatos, que facilita o trabalho das unidades de execução de instrução. Em todas as instruções de carga/armazenamento, aritméticas e lógicas, o código de operação é seguido por duas referências a registrador, com 5 bits, possibilitando referenciar 32 registradores de propósito geral.

As instruções de desvio incluem um bit de ligação (L), que indica que o endereço efetivo da instrução seguinte à instrução de desvio deve ser colocado no registrador de ligação. Duas formas de instrução também incluem um bit (A), que mostra se o modo de endereçamento é absoluto ou relativo ao PC. Nas instruções de desvio condicional, o campo CR especifica o bit do registrador de condição que deve ser testado. O campo Opções especifica as condições sob as quais o desvio deve ser efetuado. As seguintes condições podem ser especificadas:

- Desviar sempre.
- Desviar se contador for diferente de 0 e a condição for falsa.
- Desviar se contador for diferente de 0 e a condição for verdadeira.
- Desviar se contador for igual a 0 e a condição for falsa.
- Desviar se contador for igual a 0 e a condição for verdadeira.
- Desviar se contador for diferente de 0.
- Desviar se contador for igual a 0.
- Desviar se a condição for falsa.
- Desviar se a condição for verdadeira.

6 bits	5 bits	5 bits	16 bits		
Desvio	Operando imediato longo			A	L
Desvio condicional	Opções	CR	Deslocamento para desvio	A	L
Desvio condicional	Opções	CR	Deslocamento indireto, via Registrador de Ligação ou Registrador Contador		L

(a) Instruções de desvio

CR	Bit de destino	Bit de fonte	Bit de fonte	Adição, ou, ou-exclusivo etc.	/

(b) Instruções lógicas de registrador de condição

Carregar/armazenar indireto	Registrador destino	Registrador-base	Deslocamento			
Carregar/armazenar indireto	Registrador destino	Registrador-base	Registrador de índice	Tamanho, sinal, atualização	/	
Carregar/armazenar indireto	Registrador destino	Registrador-base	Deslocamento	XO		*

(c) Instruções de carga e armazenamento

Aritmética	Registrador destino	Registrador fonte	Registrador fonte	O	Adição, subtração etc.			R	
Adição, subtração etc.	Registrador destino	Registrador fonte	Valor imediato com sinal						
Lógica	Registrador fonte	Registrador destino	Registrador fonte	Adição, ou, ou-exclusivo etc.				R	
Adição, ou etc.	Registrador fonte	Registrador destino	Valor imediato com sinal						
Rotação	Registrador fonte	Registrador destino	Número de bits a serem deslocados	Início de máscara	Fim de máscara			R	
Rotação ou deslocamento	Registrador fonte	Registrador destino	Registrador fonte	Tipo de deslocamento ou máscara				R	
Rotação	Registrador fonte	Registrador destino	Número de bits a serem deslocados	Máscara		XO	S	R	*
Rotação	Registrador fonte	Registrador destino	Registrador fonte	Máscara		XO		R	*
Deslocamento	Registrador fonte	Registrador destino	Número de bits a serem deslocados	Tipo de deslocamento ou máscara			S	R	*

(d) Instruções de aritmética de números inteiros, instruções lógicas e instruções de deslocamento

Flt sgl/dbl	Registrador destino	Registrador fonte	Registrador fonte	Registrador fonte	Adição de ponto flutuante etc.	R

(e) Instruções de aritmética de ponto flutuante

A = absoluto ou relativo ao PC
* = apenas em implementações de 64 bits
L = ligação para sub-rotina
O = registra *overflow* em XER
R = registra condições em CR1
XO = extensão de código de operação
S = parte do campo de número de bits a serem deslocados

Figura 10.9 Formatos de instrução do PowerPC.

A maioria das instruções que resultam em computação (aritmética, aritmética de ponto flutuante, lógica) inclui um bit que indica se o resultado da operação deve ser anotado no registrador de condição. Como veremos, essa característica é útil no caso de processamento de previsão de desvio.

As instruções de ponto flutuante têm campos para três registradores fonte. Em muitos casos, apenas dois registradores fonte são usados. Algumas instruções envolvem a multiplicação de dois registradores fonte seguida da adição ou da subtração de um terceiro registrador fonte. A inclusão dessas instruções compostas se deve à freqüência com que são utilizadas. Por exemplo, o produto interno, que faz parte de muitas operações sobre matrizes, pode ser implementado usando a operação de multiplicação-adição.

10.5 LEITURA RECOMENDADA

As referências citadas no Capítulo 9 também se aplicam para o assunto tratado neste capítulo. Blaauw e Brooks (1997) trazem uma discussão detalhada sobre formatos de instrução e modos de endereçamento. Você pode também consultar Flynn (1985), para uma discussão e análise de questões relativas ao projeto de conjuntos de instruções, particularmente questões relacionadas a formatos.

10.6 EXERCÍCIOS

10.1 Justifique a afirmação de que uma instrução de 32 bits não é duas vezes mais útil que uma instrução de 16 bits.

10.2 Dados os seguintes valores, armazenados na memória de uma máquina com instruções de um único endereço e com um acumulador, que valores são carregados no acumulador pelas seguintes instruções?

- A palavra 20 contém o valor 40.
- A palavra 30 contém o valor 50.
- A palavra 40 contém o valor 60.
- A palavra 50 contém o valor 70.

a. CARREGA IMEDIATO 20
b. CARREGA DIRETO 20
c. CARREGA INDIRETO 20
d. CARREGA IMEDIATO 30
e. CARREGA DIRETO 30
f. CARREGA INDIRETO 30

10.3 Suponha que o endereço armazenado no contador de programa seja designado pelo símbolo X1. A instrução armazenada em X1 tem um campo de endereço (referência a operando) X2. O operando necessário para executar a instrução é armazenado na palavra de memória de endereço X3. Um registrador índice contém o valor X4. Qual é a relação entre essas várias quantidades se o modo de endereçamento da instrução é (a) direto, (b) indireto, (c) relativo ao PC e (d) indexado?

10.4 Uma instrução de desvio com modo de endereçamento relativo ao PC é armazenada na memória, no endereço 620_{10}. O desvio é feito para a posição 530_{10}. O campo de endereço da instrução tem tamanho de 10 bits. Qual é o valor binário na instrução?

10.5 Quantas vezes a CPU acessa a memória quando busca e executa uma instrução com modo de endereçamento indireto, se a instrução é (a) uma computação que requer um único operando e (b) um desvio?

10.6 O Sistema/370 da IBM não oferece endereçamento indireto. Suponha que o endereço de um operando está na memória principal. Como você faria o acesso ao operando?

10.7 Por que a decisão tomada pela IBM, de mudar o tamanho da palavra de 36 bits para 32 bits, foi um transtorno e para quem?

10.8 Em Cook (1982), o autor propõe eliminar os modos de endereçamento relativos ao PC, em favor de outros modos de endereçamento, tal como o uso de uma pilha. Qual é a desvantagem dessa proposta?

10.9 Suponha que um conjunto de instruções use um tamanho fixo de instrução de 16 bits. As referências a operandos têm tamanho de 6 bits. Existem *K* instruções com dois operandos e *L* instruções com zero operandos. Qual é o número máximo de instruções com um operando que pode ser fornecido?

10.10 Projete um código de operação com tamanho variável, de modo que permita que todas as operações a seguir sejam codificadas em uma instrução de 36 bits:

- instruções com dois endereços de 15 bits e um número de registrador de 3 bits;
- instruções com um endereço de 15 bits e um número de registrador de 3 bits;
- instruções sem endereços ou registradores.

10.11 Considere os resultados do Exercício 9.3. Suponha que M é um endereço de memória de 16 bits e que X, Y e Z podem ser endereços de 16 bits ou números de registradores de 4 bits. A máquina de um endereço usa um acumulador e as máquinas de dois ou três endereços possuem 16 registradores e instruções que operam sobre qualquer combinação de posições de memória e registradores. Supondo que os códigos de operação têm tamanho de 8 bits e que o tamanho de instruções é múltiplo de 4 bits, quantos bits cada máquina necessita para computar X?

10.12 Existe alguma justificativa possível para uma instrução com dois códigos de operação?

10.13 O Pentium II inclui a seguinte instrução:

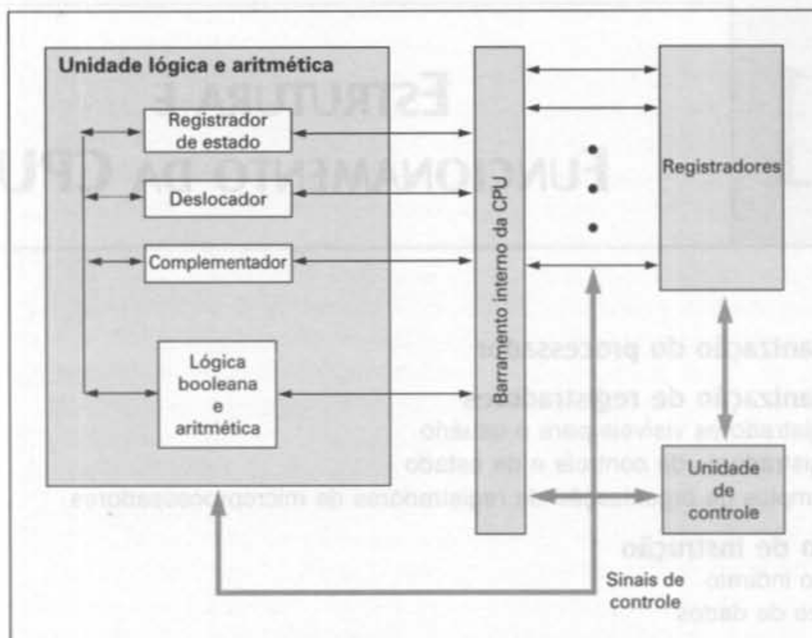
```
IMUL op1, op2, imediato
```

Essa instrução multiplica `op2`, que pode ser tanto um registrador como uma referência a operando na memória, pelo valor do operando `imediato`, e coloca o resultado em `op1`, que deve ser um registrador. Não existe nenhuma outra instrução com três operandos desse tipo no conjunto de instruções. Qual seria o possível uso dessa instrução? (*Dica*: considere a indexação.)

capítulo

11

ESTRUTURA E FUNCIONAMENTO DA CPU



- Um processador inclui tanto registradores visíveis para o usuário como registradores de controle e de estado. Os primeiros podem ser referenciados em instruções de máquina, implícita ou explicitamente, e podem ser de propósito geral ou ter uso especial, tais como armazenar números de ponto fixo ou de ponto flutuante, endereços, índices e endereços base de segmento. Os registradores de controle e de estado são usados para controlar a operação da unidade central de processamento (*central processing unit* — CPU). Um exemplo óbvio é o contador de programa. Outro exemplo importante é a palavra de estado de programa (*program status word* — PSW), que contém uma variedade de bits de estado e de condição. O PSW inclui bits que refletem o resultado da última operação aritmética executada, bits de habilitação de interrupções e um bit que indica se a CPU está executando em modo supervisor ou em modo de usuário.
- Os processadores usam a técnica de *pipelining* de instruções, para acelerar a execução. Essencialmente, a execução de instruções, na forma de uma "linha de montagem" (*pipeline*), envolve dividir o ciclo de instrução em um determinado número de estágios consecutivos, como busca de instrução, decodificação de instrução, determinação de endereço de operandos, busca de operandos, execução de instrução e escrita do resultado no operando destino. As instruções passam por meio desses estágios, assim como em uma linha de montagem, de modo que, em princípio, cada estágio possa estar trabalhando em uma instrução diferente ao mesmo tempo. A ocorrência de desvios e de dependências entre instruções complica o projeto e o uso de *pipelines*.

Este capítulo discute aspectos do processador ainda não abordados na Parte 3 e prepara a discussão sobre arquiteturas RISC e superescalares, tratada nos Capítulos 12 e 13. O capítulo começa com um resumo sobre a organização do processador. Em seguida, são analisados os registradores, que constituem a memória interna do processador. Retornamos então à discussão (iniciada na Seção 3.2) sobre o ciclo de uma instrução, para descrever uma técnica muito comum de execução de instruções, conhecida como *pipeline* de instruções. Concluímos o capítulo com o exame de alguns aspectos adicionais das organizações dos processadores Pentium II e PowerPC.

11.1 ORGANIZAÇÃO DO PROCESSADOR

Para entender a organização da CPU, devemos considerar as ações que ela deve executar:

- **Busca de instrução:** a CPU lê uma instrução da memória.
- **Interpretação de instrução:** a instrução é decodificada para determinar a ação requerida.
- **Busca de dados:** a execução de uma instrução pode requerer leitura de dados da memória ou de um módulo de E/S.
- **Processamento de dados:** a execução de uma instrução pode requerer efetuar uma operação aritmética ou lógica sobre os dados.
- **Escrita de dados:** os resultados da execução podem requerer escrever dados na memória ou em um módulo de E/S.

Para executar essas ações, a CPU precisa armazenar alguns dados temporariamente. Ela deve manter a posição de memória da última instrução, para saber onde obter a próxima instrução, e precisa também armazenar instruções e dados temporariamente, enquanto uma instrução está sendo executada. Em outras palavras, a CPU necessita de uma pequena memória interna.

A Figura 11.1 é uma visão simplificada de uma CPU, que indica também sua conexão com o resto do sistema, por meio do barramento de sistema. Uma interface semelhante é necessária para qualquer das estruturas de interconexão descritas no Capítulo 3. Você deverá se lembrar de que os componentes mais importantes da CPU são a *unidade lógica e aritmética*, ou ULA (*arithmetic and logic unit* — ALU) e a *unidade de controle*, ou UC (*control unit* — CU). A ULA efetua o processamento de dados. A unidade de controle controla não só a transferência de dados e instruções para dentro e para fora da CPU, como também a operação da ULA. A figura mostra, além desses componentes, uma memória interna mínima, constituída de um conjunto de posições de armazenamento denominadas *registradores*.

A Figura 11.2 é uma visão um pouco mais detalhada da CPU. São indicados os caminhos de transferência de dados e de sinais de controle, o que inclui um elemento denominado *barramento interno da CPU*. Esse elemento é necessário para transferir dados entre os vários registradores e a ULA, uma vez que esta última apenas opera sobre dados localizados na memória interna da CPU. A figura mostra ainda os elementos básicos típicos de uma ULA. Note a semelhança entre a estrutura interna do computador como um todo e a estrutura interna da CPU. Em ambos os casos, existe uma pequena coleção de elementos importantes (computador: CPU, E/S, memória; CPU: unidade de controle, ULA, registradores), conectados por caminhos de dados.

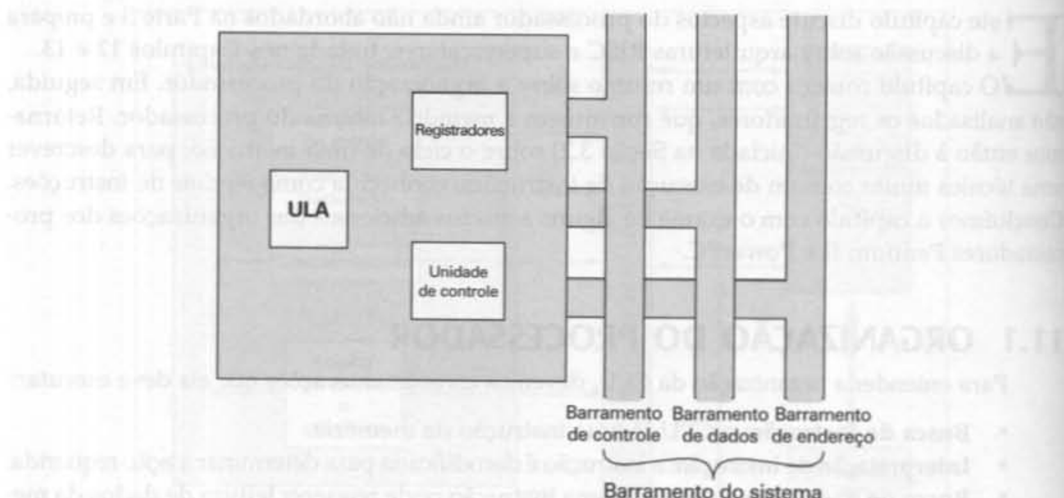


Figura 11.1 A CPU com o barramento do sistema.

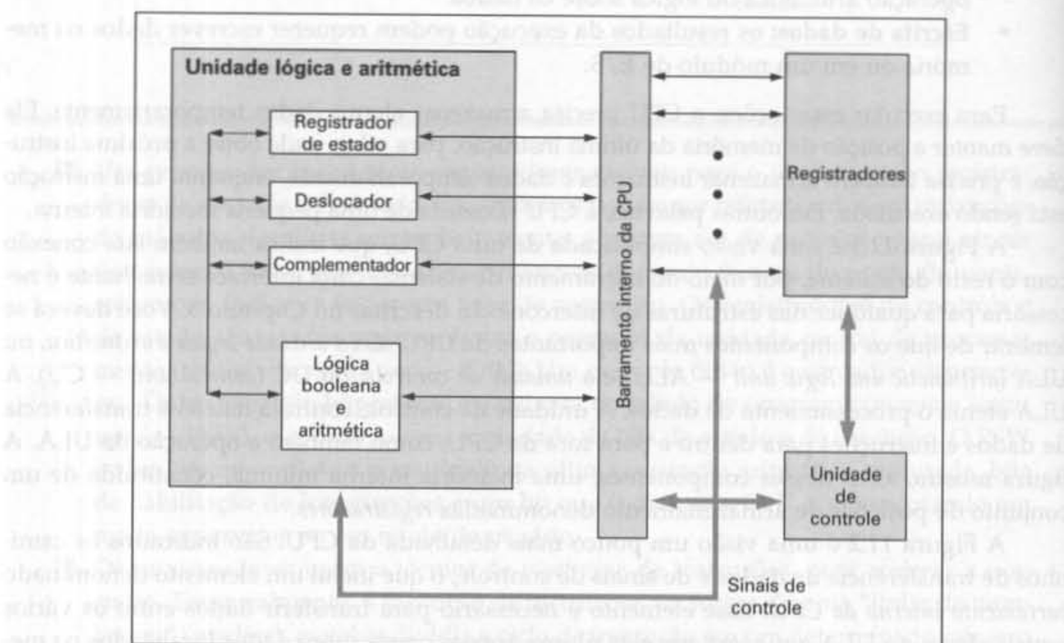


Figura 11.2 Estrutura interna da CPU.

11.2 ORGANIZAÇÃO DE REGISTRADORES

Como vimos no Capítulo 4, um sistema de computação emprega uma hierarquia de memória. Nos níveis mais altos da hierarquia, a memória é mais rápida, menor e mais cara (custo por bit). Dentro da CPU, existe um conjunto de registradores que funciona como um nível da

hierarquia de memória acima da memória principal e da memória cache. Os registradores da CPU têm duas funções:

- **Registradores visíveis para o usuário:** possibilitam ao programador de linguagem de montagem ou de máquina minimizar referências à memória, pela otimização do uso de registradores.
- **Registradores de controle e de estado:** são usados pela unidade de controle para controlar a operação da CPU e por programas privilegiados do sistema operacional para controlar a execução de programas.

Não existe uma separação clara entre os registradores dessas duas categorias. Por exemplo, em algumas máquinas, o contador de programa é visível para o usuário (no VAX), mas em muitas outras não é.

Registradores visíveis para o usuário

Um registrador visível para o usuário é aquele que pode ser referenciado pela linguagem de máquina que a CPU executa. Esses registradores podem ser classificados nas seguintes categorias:

- Registradores de propósito geral
- Registradores de dados
- Registradores de endereço
- Registradores de códigos de condição

Registradores de propósito geral podem ser usados pelo programador para uma variedade de funções. Algumas vezes, seu uso no conjunto de instruções é ortogonal ao código de operação, ou seja, qualquer registrador de propósito geral pode conter um operando para qualquer código de operação. Esse é o real significado de propósito geral. Entretanto, existem freqüentemente algumas restrições. Por exemplo, podem existir registradores dedicados para operações sobre números de ponto flutuante e para operações sobre a pilha.

Em alguns casos, os registradores de propósito geral podem ser usados para endereçamento (por exemplo, endereçamento indireto via registrador ou por deslocamento). Em outros, existe uma separação clara ou parcial entre registradores de dados e registradores de endereços. **Registradores de dados** podem ser usados apenas para conter dados e não podem ser empregados no cálculo de endereços de operandos. **Registradores de endereços** podem também ser empregados até certo ponto como registradores de propósito geral ou podem ser dedicados para um determinado modo de endereçamento. Alguns exemplos são:

- **Registradores de segmento:** em uma máquina com endereçamento segmentado (veja a Seção 7.3), um registrador de segmento é usado para conter o endereço da base de um segmento. Podem existir múltiplos registradores de segmento: por exemplo, um para o sistema operacional e um para o processo corrente.
- **Registradores de índices:** são usados para endereçamento indexado, possivelmente com auto-indexação.
- **Apontador de topo da pilha:** se houver endereçamento de operandos na pilha visível para o usuário, então tipicamente a pilha será alocada na memória e existirá um registrador dedicado que aponta para o topo da pilha. Isso possibilita um endereçamento implícito, ou seja, as instruções de empilhar e desempilhar não requerem um operando explícito.

O projeto do conjunto de registradores envolve diversas questões. Uma questão importante é decidir se os registradores serão de propósito geral ou se terão uso específico. Essa questão foi abordada no capítulo anterior, uma vez que afeta o projeto do conjunto de instruções. Com o uso de registradores especializados, o tipo de registrador referenciado como operando de uma instrução geralmente é implícito, sendo determinado pelo código de operação. O campo de operando apenas identifica um registrador de um conjunto de registradores especializados, economizando, portanto, alguns bits de instrução. Por outro lado, essa especialização limita a flexibilidade de programação. Embora não exista melhor solução para essa questão de projeto, a tendência atual é, como mencionamos anteriormente, usar registradores especializados.

Outra questão de projeto é o número de registradores a serem disponibilizados, seja para propósito geral seja para registradores de dados e de endereços. Isso também afeta o projeto do conjunto de instruções, uma vez que um número maior de registradores requer maior número de bits para especificar um operando. Como discutimos anteriormente, o número adequado parece ser entre 8 e 32 registradores (Lunde, 1977). Um pequeno número de registradores resulta em mais referências à memória, mas o uso de um número muito grande de registradores não reduz substancialmente o número de referências à memória (veja, por exemplo, Williams (1990). Entretanto, no Capítulo 12, discutiremos uma nova abordagem adotada em sistemas RISC, que obtém vantagem com a utilização de centenas de registradores.

Finalmente, existe a questão do tamanho do registrador. Registradores de endereços devem ter, obviamente, tamanho suficiente para conter o maior endereço usado no sistema. Registradores de dados devem ser capazes de conter valores da maioria dos tipos de dados. Algumas máquinas permitem o uso de dois registradores contíguos para conter valores de tamanho duplo.

Uma última categoria de registradores que são visíveis para o usuário, pelo menos parcialmente, contém **códigos de condição** (também conhecidos como *flags*). Códigos de condição são bits atualizados pelo hardware da CPU como resultado de operações. Por exemplo, em uma operação aritmética, esses bits podem indicar se o resultado produzido é positivo, negativo, zero ou *overflow*. Além de o próprio resultado da operação ser armazenado em um registrador ou na memória, são também atualizados os registradores que contêm códigos de condição. Esses códigos podem ser testados em seguida, por uma operação de desvio condicional.

Normalmente, os bits de código de condição fazem parte de um registrador de controle, embora algumas vezes possam ser organizados em mais de um registrador. As instruções de máquina geralmente possibilitam ler esses bits, por meio de uma referência implícita, mas não permitem que eles sejam alterados pelo programador.

Em algumas máquinas, uma chamada de sub-rotina provoca salvamento automático de todos os registradores visíveis para o usuário, cujos valores são restaurados no retorno da sub-rotina. A CPU salva e restaura os conteúdos desses registradores como parte da execução de instruções de chamada e retorno de sub-rotina, possibilitando que cada sub-rotina use os registradores visíveis para o usuário independentemente. Em outras máquinas, a responsabilidade de salvar os conteúdos dos registradores relevantes antes de uma chamada de sub-rotina cabe ao programador, que deve incluir instruções para esse fim no programa.

Registradores de controle e de estado

Vários registradores da CPU são empregados para controlar a operação da CPU. Na maioria das máquinas, eles não são visíveis para o usuário. Alguns deles podem ser visíveis para instruções de máquina executadas em um modo de controle ou de sistema operacional.

É claro que máquinas diferentes têm organizações de registradores diferentes e usam uma terminologia distinta. Apresentamos a seguir uma lista razoavelmente completa de tipos de registradores de controle e de estado, com uma breve descrição de cada um.

Quatro registradores são essenciais para a execução de instruções:

- **Contador de programa (PC):** contém o endereço da instrução a ser buscada.
- **Registrador de instrução (IR):** contém a última instrução buscada.
- **Registrador de endereçamento à memória (MAR):** contém o endereço de uma posição de memória.
- **Registrador de armazenamento temporário de dados (MBR):** contém uma palavra de dados a ser escrita na memória ou a palavra lida mais recentemente.

Tipicamente, o contador de programa é atualizado pela CPU depois de cada busca de instrução, de modo que ele sempre indique a próxima instrução a ser executada. Uma instrução de desvio ou de salto também modifica o conteúdo do contador de programa. A instrução buscada é carregada no IR, onde o código de operação e as referências a operando são analisadas. A troca de dados com a memória é feita usando o MAR e o MBR. Em um sistema com barramento, o MAR é conectado diretamente ao barramento de endereço e o MBR, ao barramento de dados. Registradores visíveis ao usuário, por sua vez, trocam dados com o MBR.

Os quatro registradores relacionados anteriormente são usados para transferência de dados entre a CPU e a memória. Dentro da CPU, os dados devem ser apresentados à ULA para processamento. A ULA pode ter acesso direto ao MBR e aos registradores visíveis para o usuário. Alternativamente, podem existir registradores adicionais para armazenamento temporário de dados, que servem como registradores de entrada e de saída da ULA e trocam dados com o MBR e os registradores visíveis para o usuário.

Todo projeto da CPU inclui um registrador, ou conjunto de registradores, freqüentemente conhecido como *palavra de estado de programa* (PSW), que contém informação de estado. Tipicamente, o registrador PSW contém códigos de condição e outras informações de estado, incluindo os seguintes campos:

- **Sinal:** contém o bit de sinal do resultado da última operação aritmética.
- **Zero:** atualizado com valor 1 se o resultado da última operação for 0.
- **'Vai-um':** atualizado com valor 1 se uma operação resultar em um 'vai-um' para fora do bit de ordem superior (adição) ou em um 'vem-um' para o bit de ordem superior (subtração). É usado por operações aritméticas de múltiplas palavras.
- **Igual:** atualizado com valor 1 se uma comparação lógica resultar em igualdade.
- ***Overflow*:** usado para indicar *overflow* aritmético.
- **Habilitar/desabilitar interrupção:** usada para habilitar ou desabilitar interrupções.
- **Supervisor:** indica se a CPU está executando em modo supervisor ou em modo de usuário. Certas instruções privilegiadas apenas podem ser executadas no modo supervisor, assim como certas áreas de memória apenas podem ser acessadas no modo supervisor.

O projeto de uma CPU pode também incluir outros registradores relacionados ao estado e ao controle. Além do registrador PSW, deve existir um registrador que aponta para um bloco de memória que contém informação de estado adicional (por exemplo, blocos de controle de processo). Em máquinas que usam vetor de interrupções, pode existir um registrador de vetor de interrupções. Se for usada uma pilha para implementar certas funções (por exemplo, chamada de sub-rotina), será necessário um registrador indicador de topo de pilha. Em um sistema com memória virtual, um registrador é usado para apontador para a tabela de páginas. Finalmente, podem também ser usados registradores para o controle de operações de E/S.

Diversos fatores devem ser considerados no projeto da organização de registradores de controle e de estado. Uma questão-chave é o suporte para o sistema operacional. Certos tipos de informação de controle são úteis especificamente para o sistema operacional. Se o projetista da CPU tem entendimento funcional sobre o sistema operacional a ser usado, a organização dos registradores pode ser feita razoavelmente de acordo com o sistema operacional.

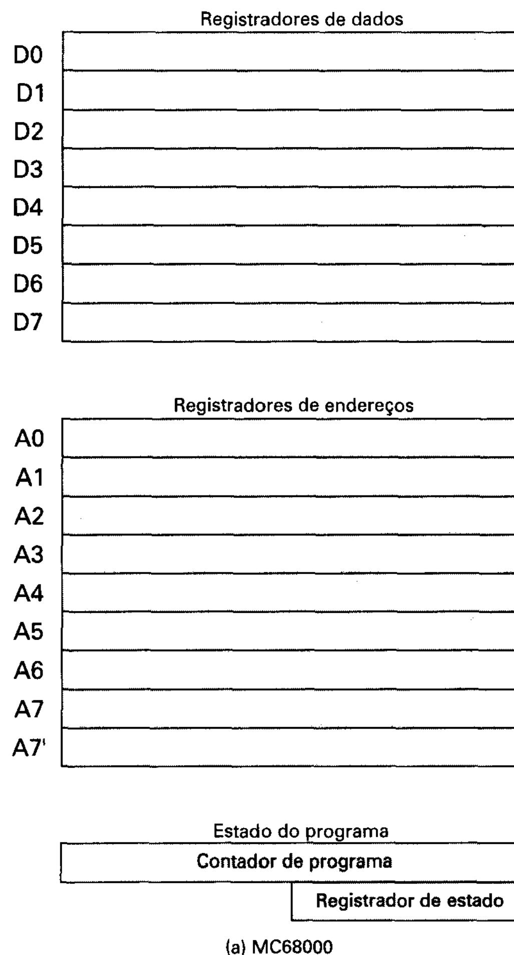
Outra decisão de projeto importante é a alocação de informação de controle entre registradores e memória. É comum reservar as primeiras (de endereço mais baixo) centenas ou milhares de palavras da memória para armazenar informações de controle. O projetista deve decidir que parte das informações de controle deve ser mantida em registradores e que parte deve ficar na memória, levando-se em conta o custo e a velocidade de acesso.

Exemplos de organização de registradores de microprocessadores

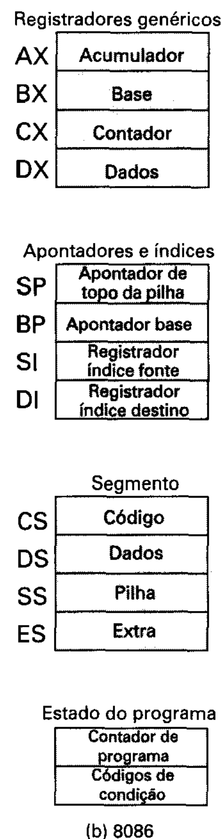
É instrutivo examinar e comparar organizações de registradores de sistemas similares. Nesta seção, abordamos dois microprocessadores de 16 bits que foram projetados quase ao mesmo tempo: o Motorola MC68000 (Stritter, 1979) e o Intel 8086 (Morse, 1978). As Figuras 11.3a e 11.3b representam a organização de registradores de cada um; os registradores usados apenas internamente pela CPU, como registradores de endereçamento à memória, não são mostrados.

O MC68000 divide seus registradores de 32 bits em oito registradores de dados e nove registradores de endereço. Os oito registradores de dados são usados principalmente para manipulação de dados, mas também como registradores índice. O tamanho dos registradores permite operar dados de 8, 16 ou 32 bits, conforme determinado pelo código de operação. Os registradores de endereço contêm endereços de 32 bits (sem segmentação); dois desses registradores são também usados como apontadores de pilha, um para o usuário e outro para o sistema operacional, dependendo do modo de execução corrente. Esses dois registradores são ambos identificados pelo número 7, uma vez que apenas um pode ser usado de cada vez. O MC68000 inclui também um contador de programa de 32 bits e um registrador de estado de 16 bits.

Os projetistas da Motorola tinham como objetivo definir um conjunto de instruções bastante regular, sem registradores de uso especial. A preocupação com a eficiência de código levou-os a dividir os registradores em dois grupos funcionais, economizando um bit na especificação de registradores e mantendo, assim, um compromisso razoável entre a total generalidade e a compactação de código.



(a) MC68000



(b) 8086



(c) 80386 – Pentium II

Figura 11.3 Exemplos de organizações de registradores de microprocessadores.

O Intel 8086 adota uma abordagem diferente para a organização dos registradores. Todo registrador é de uso especial, embora alguns também possam ser de propósito geral. O 8086 contém quatro registradores de dados de 16 bits, que podem ser endereçados em unidades de 16 bits ou byte a byte, e quatro outros registradores de 16 bits, usados como apontadores e registradores índice. Os registradores de dados podem ser usados como propósito geral em algumas instruções, sendo, em outras, usados implicitamente. Por exemplo, uma instrução de multiplicação usa sempre o acumulador. Os quatro registradores apontadores são também usados implicitamente em diversas operações; cada um contém um deslocamento relativo ao início de um segmento. Existem também quatro registradores de segmento de 16 bits, dos quais três são usados de modo dedicado e implícito, para indicar o segmento da instrução corrente (útil para instruções de desvio), um segmento de dados e um segmento de pilha. O uso desses registradores de modo dedicado e implícito proporciona uma codificação mais compacta, ao custo de uma flexibilidade mais reduzida. O 8086 inclui também um contador de programa e um conjunto de bits de estado e de controle.

O objetivo dessa comparação deve ficar claro. Não existe, até o momento, uma filosofia sobre a melhor maneira de organizar os registradores da CPU que seja universalmente aceita (Toong, 1981). Assim como no projeto do conjunto de instruções, e em muitas outras questões de projeto da CPU, as decisões são baseadas em critérios e preferências.

Outro ponto instrutivo a respeito do projeto da organização de registradores é mostrado na Figura 11.3c. Essa figura apresenta a organização dos registradores visíveis para o usuário no Intel 80386 (El-Ayat, 1985), que é um microprocessador de 32 bits projetado como extensão do 8086[1]. O 80386 usa registradores de 32 bits. Entretanto, para fornecer compatibilidade com programas escritos para máquinas anteriores, ele retém a organização de registradores original embutida na nova organização. Em virtude dessa restrição, os projetistas dos processadores de 32 bits tinham flexibilidade limitada para definir a organização de registradores.

11.3 CICLO DE INSTRUÇÃO

Na Seção 3.2, descrevemos o ciclo de instrução da CPU. A Figura 11.4 repete uma das figuras usadas nessa descrição (Figura 3.9). Relembrando, o ciclo de instrução inclui os seguintes subciclos:

- **Busca:** lê a próxima instrução da memória para a CPU.
- **Execução:** interpreta o código de operação e efetua a operação indicada.
- **Interrupção:** se as interrupções estão habilitadas e ocorreu uma interrupção, salva o estado do processo atual e processa a interrupção.

Podemos agora detalhar um pouco mais o ciclo de instrução. Primeiramente, devemos introduzir um subciclo adicional, conhecido como ciclo indireto.

1. Como o MC68000 já usa registradores de 32 bits, o MC68020 (MacGregor, 1984), uma extensão desse processador que possui caminhos de dados e endereços com 32 bits, mantém a mesma organização de registradores.

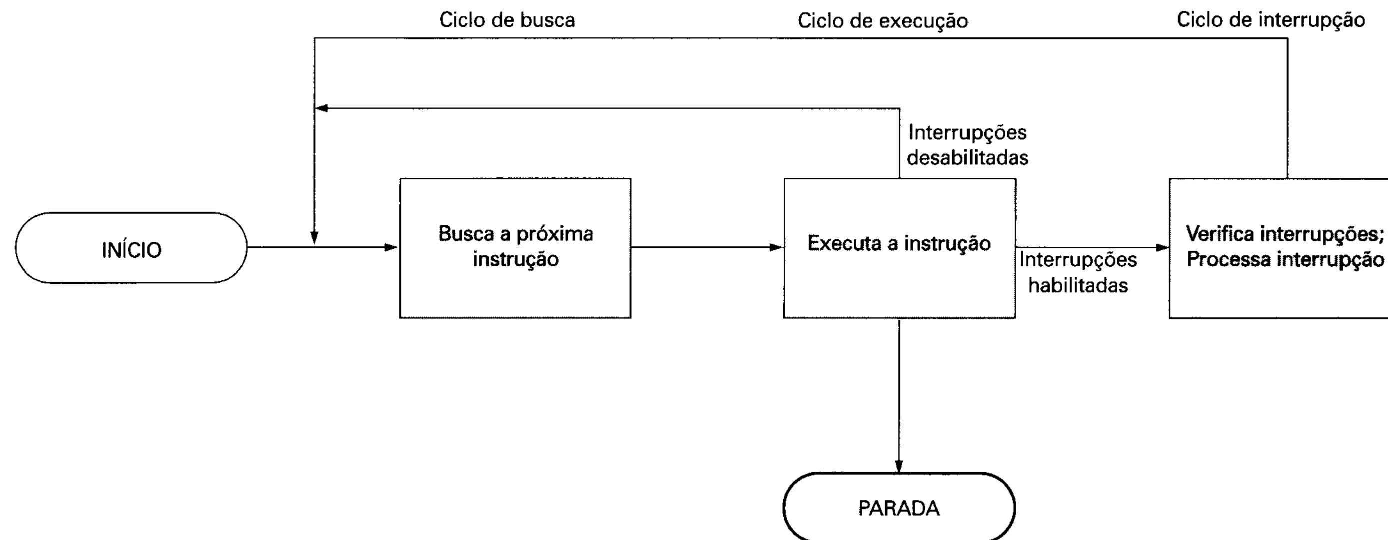


Figura 11.4 Ciclo de instrução com interrupções.

Ciclo indireto

No Capítulo 10, vimos que a execução de uma instrução pode envolver um ou mais operandos, cada um dos quais requerendo um acesso à memória. Além disso, se for usado endereçamento indireto, serão requeridos acessos à memória adicionais.

Podemos imaginar a busca de endereços indiretos como mais um subciclo de instrução. O resultado é mostrado na Figura 11.5. A linha principal de atividade consiste em alternar as atividades de busca de instruções e de execução de instrução. Depois que uma instrução é buscada, ela é examinada para determinar se algum endereçamento indireto está envolvido. Se isso ocorrer, os operandos requisitados são buscados, usando endereçamento indireto. Seguindo a execução, pode ocorrer processamento de uma interrupção, antes da busca da próxima instrução.

Outra maneira de ver esse processo é apresentada na Figura 11.6, que é uma versão revisada da Figura 3.12. Essa figura mostra mais corretamente a natureza do ciclo de instrução. Quando uma instrução é buscada, seus campos de referência a operandos devem ser identificados. Cada operando de entrada localizado na memória é então buscado, podendo esse processo requerer endereçamento indireto. Operandos localizados em registradores não precisam ser buscados. Depois que a operação é executada, pode ser requerido um processo semelhante para armazenar o resultado na memória.

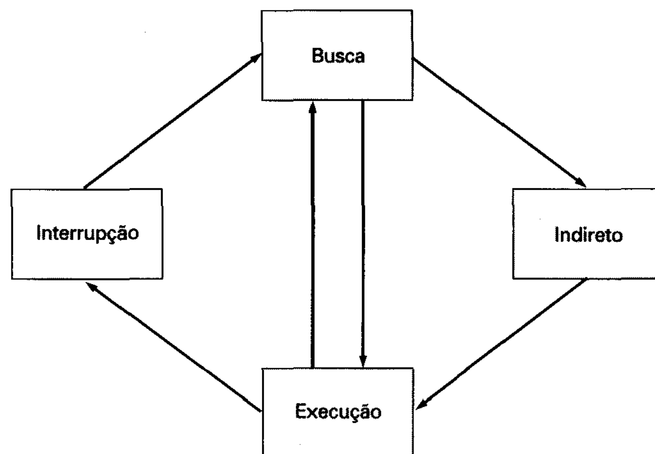


Figura 11.5 Ciclo de instrução.

Fluxo de dados

A seqüência exata de eventos durante um ciclo de instrução depende do projeto da CPU. É possível, entretanto, indicar o que pode acontecer em termos gerais. Suponha que a CPU empregue um registrador de endereço de memória (MAR), um registrador de armazenamento temporário de dados (MBR), um contador de programa (PC) e um registrador de instruções (IR).

Durante o *ciclo de busca*, uma instrução é lida a partir da memória. A Figura 11.7 mostra o fluxo de dados durante esse ciclo. O contador de programa (PC) contém o endereço da próxima instrução a ser buscada. Esse endereço é movido para o MAR e colocado no barramento de endereço. A unidade de controle requisita uma leitura na memória, e o resultado é colocado no barramento de dados e copiado no MBR e então movido para o IR. Enquanto isso, o contador de programa é incrementado de 1, para preparar a próxima busca de instrução.

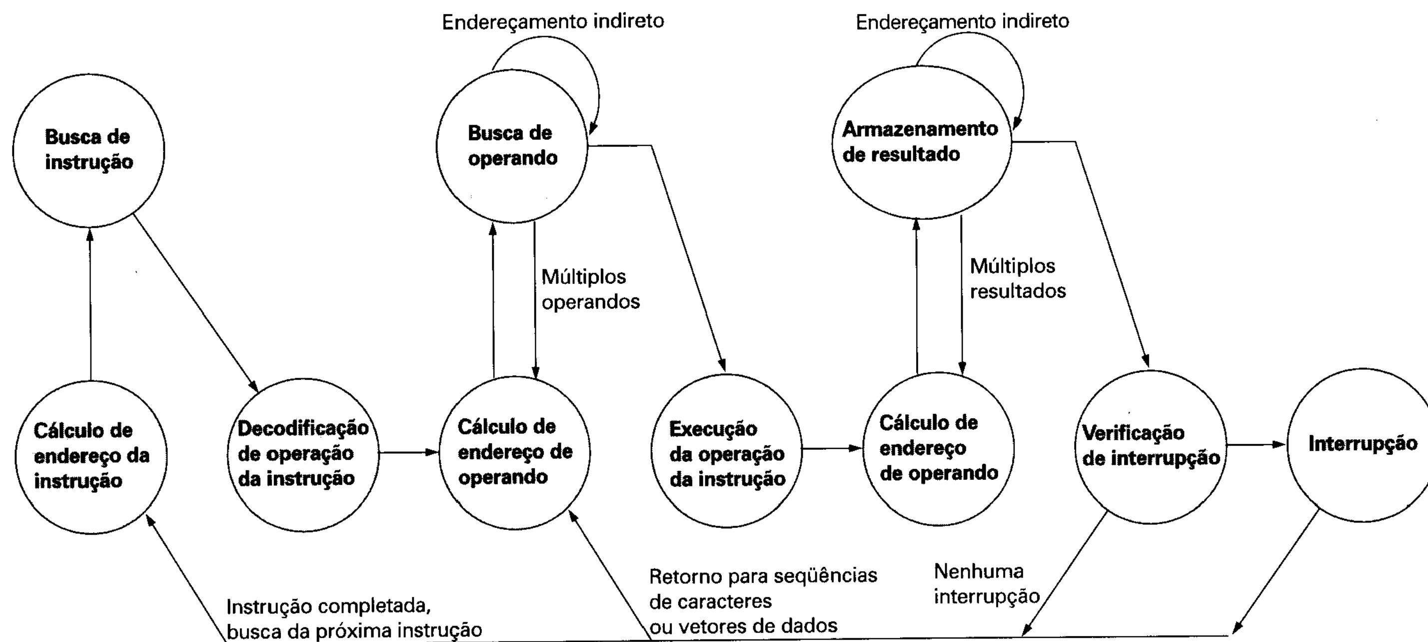
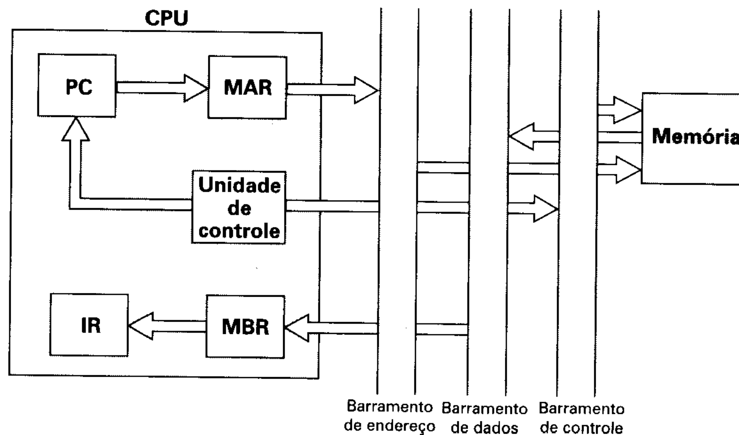


Figura 11.6 Diagrama de transição de estados do ciclo de instrução.



MBR = Registrador de armazenamento temporário de dados

MAR = Registrador de endereço de memória

IR = Registrador de instruções

PC = Contador de programa

Figura 11.7 Fluxo de dados no ciclo de busca.

Uma vez que o ciclo de busca termina, a unidade de controle examina o conteúdo do IR para determinar se a instrução especifica algum operando com endereçamento indireto. Se isso ocorrer, um *ciclo indireto* é efetuado. Como mostra a Figura 11.8, esse ciclo é bastante simples. Os *N* bits mais à direita do MBR, que contém a referência ao endereço, são transferidos para o MAR. Então, a unidade de controle requisita uma leitura na memória, para transferir o endereço do operando desejado para o MBR.

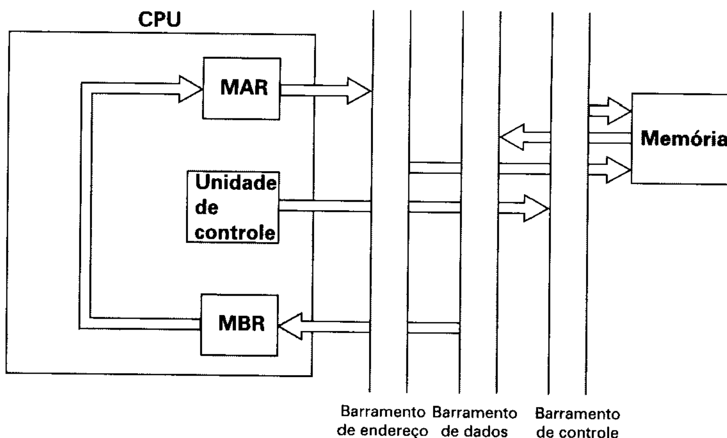


Figura 11.8 Fluxo de dados no ciclo indireto.

O ciclo de busca e o ciclo indireto são simples e previsíveis. O *ciclo de execução* pode ter muitas formas, dependendo de qual das várias instruções de máquina está contida no IR. Ele

pode envolver transferência de dados entre registradores, leitura e escrita na memória ou em dispositivos de E/S e invocação da ULA.

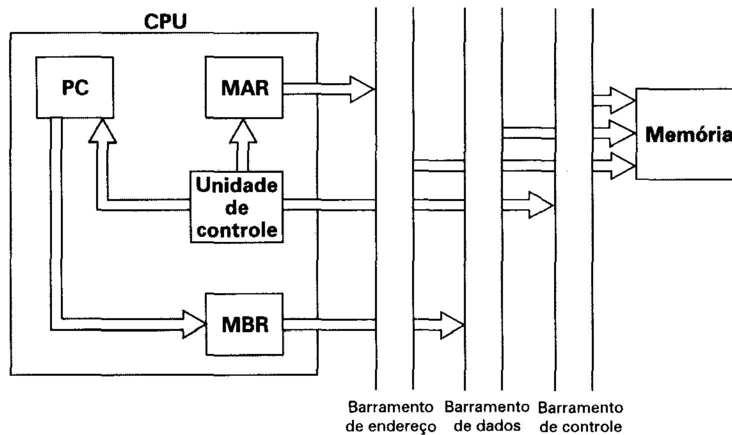


Figura 11.9 Fluxo de dados no ciclo de interrupção.

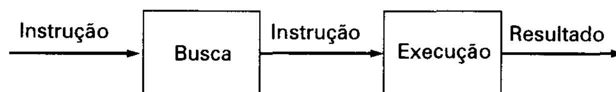
Assim como o ciclo de busca e o ciclo indireto, o *ciclo de interrupção* é simples e previsível (Figura 11.9). O conteúdo corrente do PC deve ser salvo, para que a CPU possa retomar sua atividade normal depois de processar a interrupção. Assim, o conteúdo do PC é transferido para o MBR, para depois ser escrito na memória. A posição especial da memória reservada para esse propósito é carregada no MAR pela unidade de controle. Ela pode ser, por exemplo, a posição apontada pelo registrador de topo da pilha. O PC é carregado com o endereço da rotina de interrupção. Como resultado, o próximo ciclo de instrução começará buscando a instrução apropriada.

11.4 *PIPELINE* DE INSTRUÇÕES

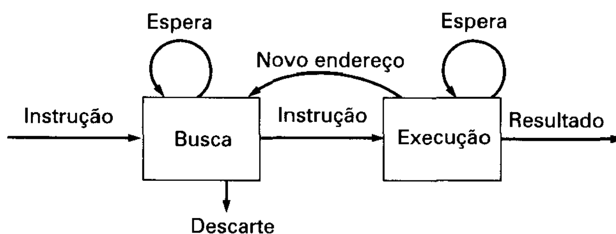
À medida que os sistemas de computação evoluem, é possível obter maior desempenho com o uso de tecnologias mais avançadas, tais como um conjunto de circuitos mais rápidos. Além disso, uma melhor organização da CPU pode também melhorar o desempenho. Alguns exemplos disso foram vistos anteriormente, tais como o uso de múltiplos registradores no lugar de um único acumulador e o uso de memória cache. Outra abordagem comum na organização da CPU é o uso de uma *pipeline* de instruções.

Estratégia de *pipeline*

Uma *pipeline* de instruções é semelhante a uma linha de montagem de uma indústria. Uma linha de montagem tira proveito do fato de que um produto passa por vários estágios de produção: produtos em vários estágios do processo de produção podem ser trabalhados simultaneamente. Em uma *pipeline* de instruções, assim como em uma linha de montagem, novas entradas são aceitas em uma extremidade, antes que entradas aceitas previamente apareçam como saídas na outra extremidade.



(a) Visão simplificada



(b) Visão expandida

Figura 11.10 *Pipeline* de instruções de dois estágios.

Para aplicar esse conceito para a execução de instruções, precisamos reconhecer que, de fato, uma instrução possui vários estágios. A Figura 11.6, por exemplo, divide o ciclo de instrução em dez tarefas que ocorrem em seqüência. De maneira clara, existe oportunidade para trabalhar simultaneamente várias instruções, cada uma em um diferente estágio de execução.

Como uma abordagem mais simples, suponha que o processamento de uma instrução é subdividido em dois estágios: busca da instrução e execução da instrução. Existem momentos durante a execução de uma instrução em que a memória principal não está sendo usada. Esse instante pode ser usado para buscar a próxima instrução, em paralelo com a execução da instrução corrente. A Figura 11.10a representa essa abordagem. A *pipeline* tem dois estágios independentes. O estágio busca uma instrução e a armazena em uma área de armazenamento temporário. Quando o segundo estágio está livre, o primeiro passa para ele a instrução armazenada. Enquanto o segundo está executando essa instrução, o primeiro tira proveito de ciclos de memória que não são usados para buscar e armazenar a próxima instrução. Isso é chamado de *busca antecipada de instrução* (*instruction prefetch*) ou *superposição de busca* (*fetch overlap*).

Note que esse processo acelera a execução de instruções. Se os estágios de busca e de execução tiverem a mesma duração, o número de instruções executadas por unidade de tempo será dobrado. Entretanto, se examinarmos bem essa *pipeline* (Figura 11.10b), veremos que essa duplicação da taxa de execução de instruções será pouco provável, por duas razões:

1. O tempo de execução geralmente é maior que o tempo de busca, pois a execução de uma instrução geralmente envolve leitura e armazenamento de operandos e execução de algumas operações. Portanto, o estágio de busca pode ter de esperar algum tempo antes que possa esvaziar sua área de armazenamento temporário.
2. A ocorrência de instruções de desvio condicional faz com que o endereço da próxima instrução a ser buscada seja desconhecido. Nesse caso, o estágio de busca teria de esperar até receber o endereço da próxima instrução do estágio de execução. O estágio de execução poderia, então, ter de esperar enquanto a próxima instrução é buscada.

Em razão da ocorrência de instruções de desvio, o tempo perdido pode ser reduzido pelo uso de uma estratégia de adivinhação. Uma regra simples é a seguinte: quando uma ins-

trução de desvio condicional é passada do estágio de busca para o de execução, o estágio de busca obtém na memória a instrução imediatamente seguinte à instrução de desvio. Então, se não ocorrer o desvio, nenhum tempo será perdido. Se ocorrer o desvio, a instrução buscada deve ser descartada, sendo buscada uma nova instrução.

Embora esses fatores reduzam a potencial efetividade da *pipeline* de dois estágios, algum ganho de desempenho é obtido. Para conseguir maior desempenho, a *pipeline* deve ter maior número de estágios. Considere a seguinte decomposição do processamento de uma instrução:

- **Busca de instrução (BI):** lê a próxima instrução esperada e a armazena em uma área de armazenamento temporário.
- **Decodificação da instrução (DI):** determina o código de operação da instrução e as referências a operandos.
- **Cálculo de operandos (CO):** determina o endereço efetivo de cada operando fonte. Isso pode envolver endereçamento por deslocamento, endereçamento indireto via registrador, endereçamento indireto, assim como outras formas de cálculo de endereço.
- **Busca de operandos (BO):** busca cada operando localizado na memória. Os operandos localizados em registradores não precisam ser buscados.
- **Execução da instrução (EI):** efetua a operação indicada e armazena o resultado, se houver, na localização do operando de destino especificado.
- **Escrita de operando (EO):** armazena o resultado na memória.

Com essa decomposição, os vários estágios têm duração aproximadamente igual. Para fins de ilustração, vamos supor que a duração de cada estágio seja igual. A Figura 11.11 mostra que uma *pipeline* de seis estágios pode reduzir o tempo de execução de 9 instruções de 54 para 14 unidades de tempo.

Diversos comentários são importantes nesse ponto. O diagrama considera que cada instrução passa por todos os seis estágios da *pipeline*. Nem sempre isso acontece. Por exemplo, uma instrução de carga não necessita do estágio EO. Entretanto, para simplificar o hardware da *pipeline*, a sincronização é feita ao supor que cada instrução requer todos os seis estágios. Além disso, o diagrama supõe que todos os estágios possam ser executados em paralelo. Em particular, supõe-se que não existam conflitos de acesso à memória. Por exemplo, os estágios BI, BO e EO envolvem acesso à memória. O diagrama subentende que a memória pode ser usada simultaneamente por esses estágios, o que não é possível na maioria dos sistemas de memória. Entretanto, o valor desejado pode estar na memória cache ou os estágios BO e EO podem não ser executados. Assim, esses conflitos de acesso à memória muitas vezes não diminuem a velocidade de execução de instruções na *pipeline*.

Diversos outros fatores limitam o aumento de desempenho. Se os seis estágios não têm duração igual, existe certa espera envolvida em vários estágios da *pipeline*, como foi discutido anteriormente para a *pipeline* de dois estágios. Outra dificuldade é que uma instrução de desvio condicional pode invalidar diversas buscas de instrução. Um evento imprevisível semelhante é a ocorrência de uma interrupção. A Figura 11.12 mostra o efeito de um desvio condicional, que usa o mesmo programa da Figura 11.11. Suponha que a instrução 3 seja um desvio condicional para a instrução 15. Até que a execução dessa instrução seja efetuada, não há como saber qual a instrução que virá a seguir. Nesse exemplo, a *pipeline* simplesmente carrega a próxima instrução da seqüência (instrução 4) e prossegue a execução. Na Figura 11.11, o desvio não é tomado e, portanto, obtemos total benefício do aumento de desempenho. Na

Figura 11.12, o desvio é tomado, mas isso é determinado apenas ao final da unidade de tempo 7. Nesse ponto, devem ser retiradas da *pipeline* as instruções que não são úteis. Durante a unidade de tempo 8, a instrução 15 entra na *pipeline*. Nenhuma instrução é completada durante as unidades de tempo 9 a 12; essa é a penalidade de desempenho em virtude do fato de não termos previsto corretamente o resultado da instrução de desvio. A Figura 11.13 mostra a lógica necessária para que a *pipeline* possa tratar desvios e interrupções.

O uso de uma *pipeline* de seis estágios apresenta alguns problemas que não ocorrem na organização mais simples de dois estágios. O estágio CO pode depender do conteúdo de um registrador que pode ser alterado por uma instrução anterior que ainda está na *pipeline*. Outros conflitos de memória e de registradores podem ocorrer. O sistema deve conter algum tipo de lógica para tratar esses conflitos.

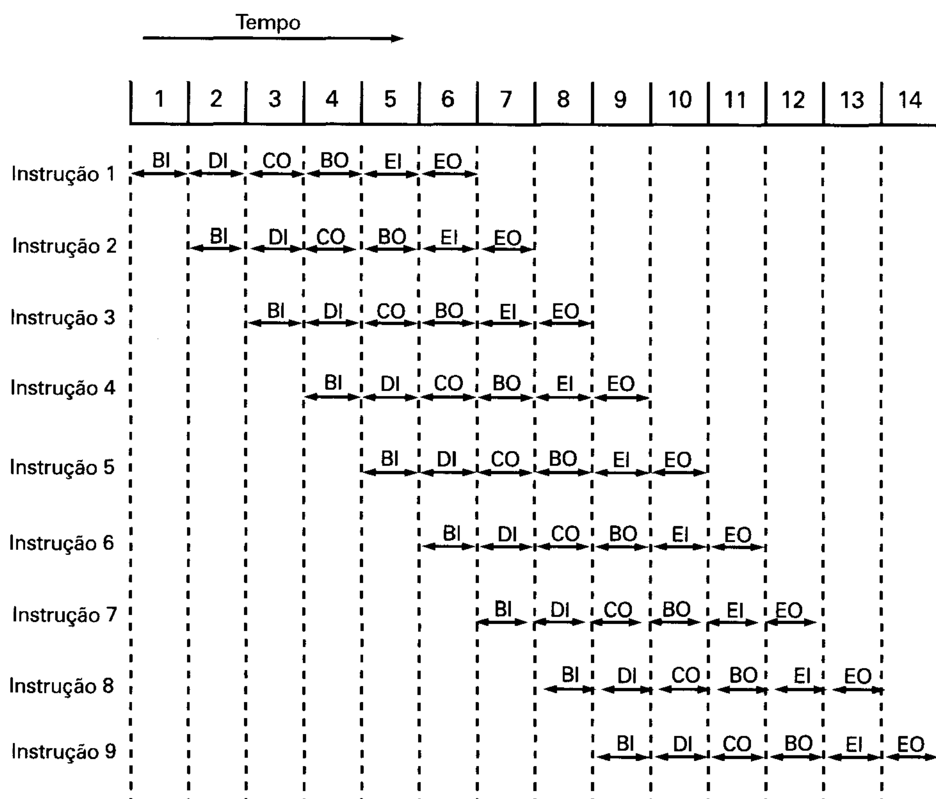


Figura 11.11 Diagrama de tempo para operação da *pipeline* de instruções.

Da discussão precedente, pode parecer que quanto maior é o número de estágios na *pipeline*, mais rápida é a taxa de execução. Alguns projetistas do S/360 da IBM apontaram dois fatores que frustram esse padrão aparentemente simples para projetos de alto desempenho (Anderson, 1967) e que permanecem válidos até hoje:

1. Em cada estágio da *pipeline*, existe certo custo (*overhead*) envolvido na movimentação de dados entre áreas de armazenamento temporário e na execução de várias atividades de preparação e entrega de dados. Esse custo pode aumentar consideravelmente o tempo total de execução de uma única instrução. Isso é significativo quando instruções consecutivas são logicamente dependentes, seja pelo uso intensivo de desvios, seja por dependências de acesso à memória.
2. A lógica de controle requerida para manipular dependências entre acessos à memória ou entre registradores, assim como para otimizar o uso da *pipeline*, aumenta bastante com o número de estágios. Isso pode levar a uma situação em que a lógica que controla a comutação entre estágios é mais complexa que os estágios a serem controlados.

O uso da *pipeline* de instruções é uma técnica poderosa para aumentar o desempenho, mas requer um projeto cuidadoso para que possa alcançar resultados ótimos com uma complexidade razoável.

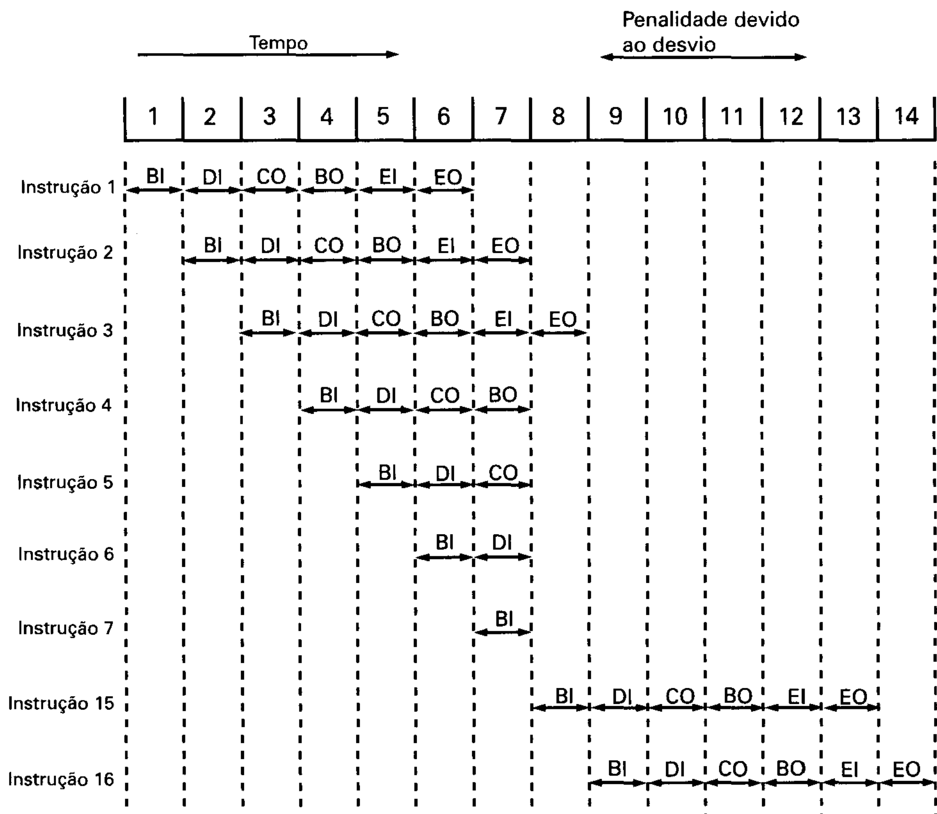


Figura 11.12 O efeito de um desvio condicional na operação de uma *pipeline* de instruções.

Desempenho da *pipeline*

Nesta subseção, derivamos algumas medidas simples do desempenho e do *speedup* relativo do processo de execução de instruções com o uso de *pipeline* (com base na discussão apresentada em Hwang, 1993). O tempo de ciclo τ de uma *pipeline* de instrução é o tempo requerido para avançar um conjunto de instruções um estágio por meio da *pipeline*; cada coluna nas Figuras 11.11 e 11.12 representa o tempo de um ciclo. O tempo de ciclo pode ser determinado da seguinte maneira:

$$\tau = \max[\tau_i] + d = \tau_m + d \qquad i, 1 \le i \le k$$

onde:

τ_m = atraso máximo de estágio (atraso por meio do estágio de maior atraso)

k = número de estágios da *pipeline* de instrução

d = tempo necessário para propagar sinais e dados de um estágio para o próximo

Em geral, o tempo de atraso d é equivalente ao pulso de um relógio e $\tau m >> d$. Suponha agora que sejam processadas n instruções, sem que ocorra desvio. O tempo total T_k requerido para executar as n instruções é:

$$T_k = [k + (n - 1)]\tau \qquad (11.1)$$

Um total de k ciclos é necessário para completar a execução da primeira instrução e as $n - 1$ instruções restantes requerem $n - 1$ ciclos[2]. Essa equação é facilmente verificada por meio da Figura 11.11. A nona instrução é completada no ciclo 14:

$$14 = [6 + (9 - 1)]$$

O *speedup* para a execução com a *pipeline* de instruções em relação à execução sem o uso da *pipeline* é definida como:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \qquad (11.2)$$

A Figura 11.14a mostra um gráfico do *speedup* de execução de instruções em função do número de instruções executadas sem que ocorra um desvio. Como se poderia esperar, temos que, no limite ($n \rightarrow \infty$), o fator de aceleração é igual a k. A Figura 11.14b mostra um gráfico do *speedup* de execução de instruções em função do número de estágios da *pipeline*[3]. Nesse caso, o fator de aceleração se aproxima do número de instruções que podem ser introduzidas na *pipeline* sem que ocorra desvio. Portanto, quanto maior o número de estágios da *pipeline*, maior o *speedup*. Entretanto, na prática, o ganho potencial com a introdução de estágios adicionais na *pipeline* é diminuído pelo aumento de custo, por atrasos entre estágios e pelo fato de que instruções de desvio requerem que a *pipeline* seja esvaziada. O uso de um número de estágios entre 6 e 9 parece ser mais adequado, sendo contraproducente o uso de um número de estágios maior.

2. Estamos sendo um pouco superficiais aqui. Esse número de ciclos é apenas aproximado. O tempo de ciclo somente será igual ao valor máximo de τ quando todos os estágios estiverem cheios. Para os primeiros ciclos, o tempo de ciclo pode ser menor.

3. Note que, na Figura 11.14a, a escala usada no eixo x é logarítmica, enquanto na Figura 11.14b ela é linear.

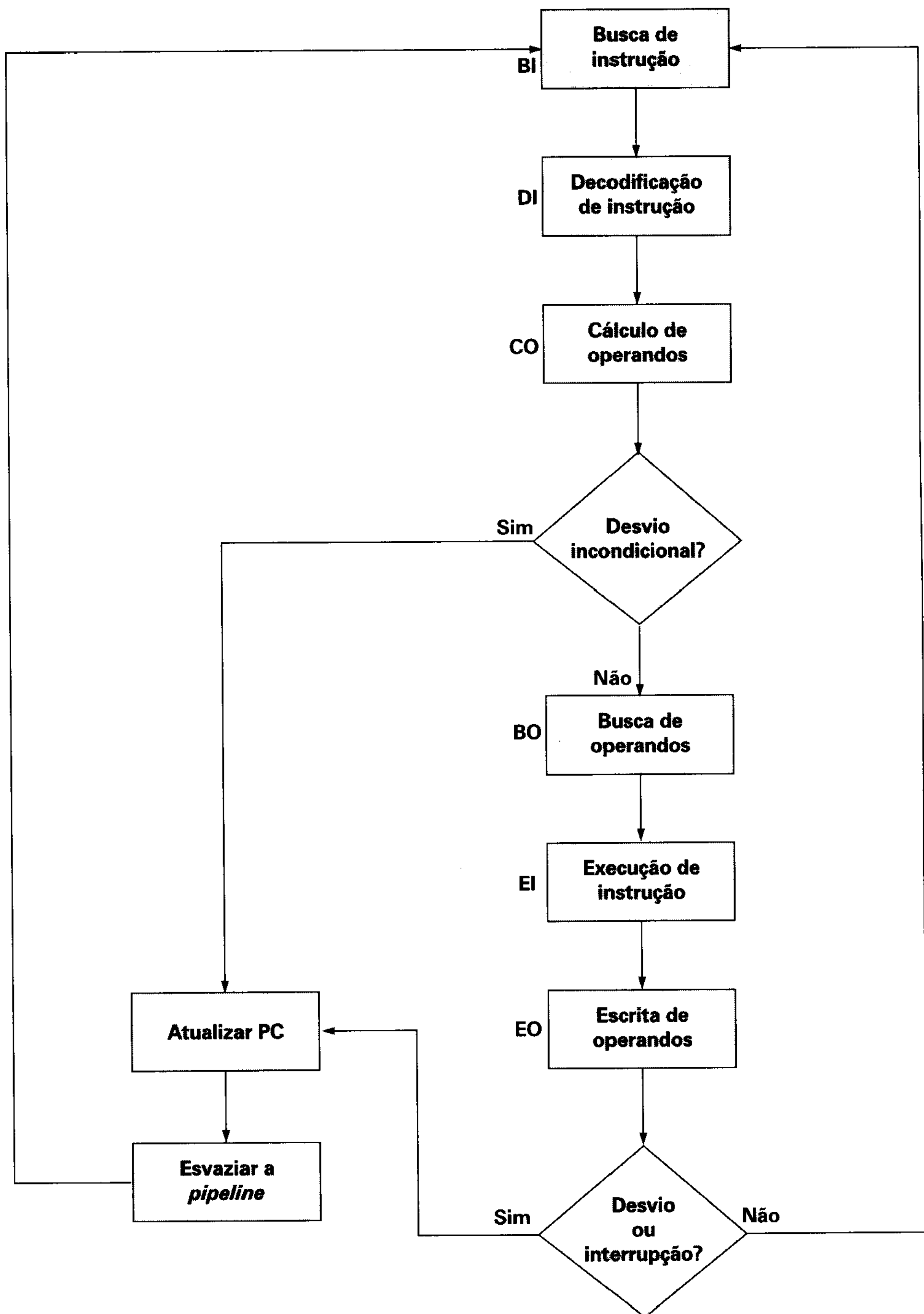


Figura 11.13 *Pipeline* de instruções de seis estágios.

Lidando com desvios

Um dos maiores problemas do projeto de uma *pipeline* de instruções é assegurar um fluxo constante de instruções nos estágios iniciais da *pipeline*. O principal impedimento a isso é, como vimos, a existência de instruções de desvio condicional. Antes que a instrução seja realmente executada, é impossível determinar se o desvio será tomado ou não.

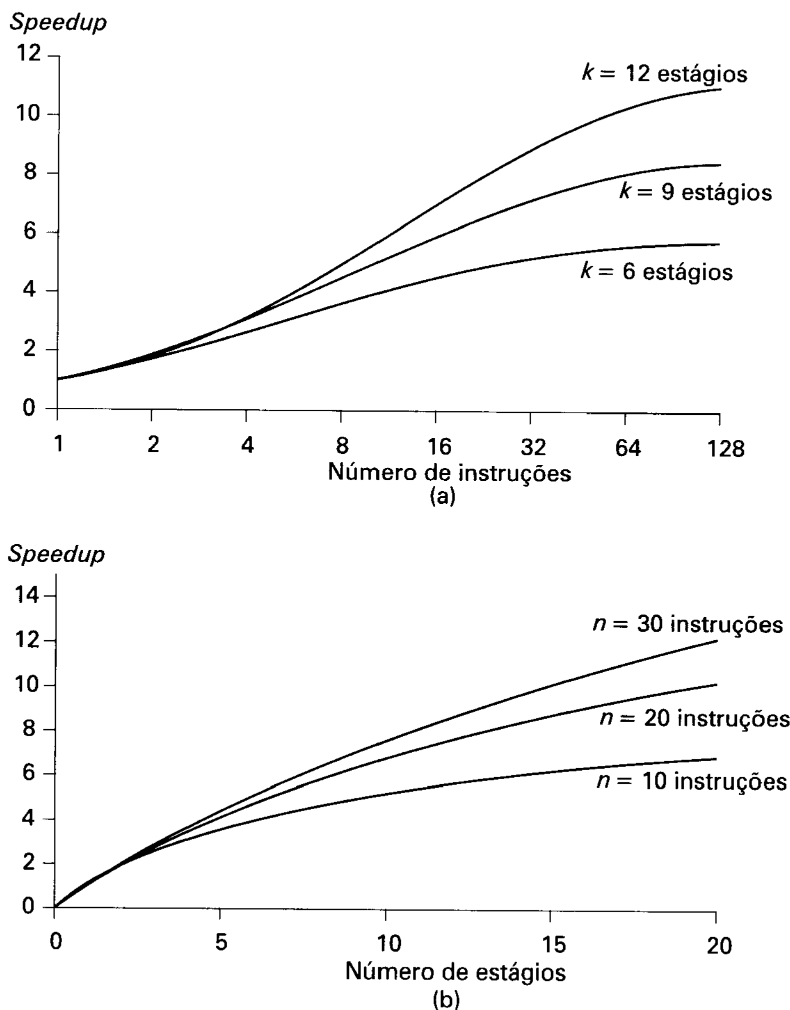


Figura 11.14 *Speedups* obtidos com o uso de *pipeline* de instruções.

Diversas abordagens são adotadas para lidar com desvios condicionais:

- Múltiplos fluxos
- Antecipação de busca da instrução-alvo de desvio
- Memória de laço de repetição
- Previsão de desvio
- Atraso do desvio (*delayed branch*)

Múltiplos fluxos

Uma *pipeline* simples experimenta certa penalidade, em razão de instruções de desvio, porque pode fazer uma escolha errada entre as duas instruções possíveis para ser a próxima instrução a ser buscada. Uma possível abordagem (do tipo 'força bruta') para resolver esse problema consiste em duplicar os estágios iniciais da *pipeline* para permitir a busca de ambas as instruções, usando assim dois fluxos de instruções. Essa abordagem apresenta dois problemas:

- O uso de múltiplas *pipelines* introduz atrasos devidos à contenção de acesso a registradores e à memória.
- Pode ocorrer a entrada de instruções de desvio adicionais na *pipeline* (em qualquer um dos fluxos), antes que seja tomada a decisão sobre o desvio original. Cada nova instrução de desvio introduzida na *pipeline* requer um fluxo de instruções adicional.

Apesar dessas desvantagens, essa estratégia pode melhorar o desempenho. Exemplos de máquinas que utilizam dois ou mais fluxos de *pipeline* são o 370/168 e o 3033, ambos da IBM.

Busca antecipada da instrução-alvo de desvio

Essa técnica consiste em buscar antecipadamente tanto a instrução-alvo do desvio quanto a instrução consecutiva ao desvio, no instante em que uma instrução de desvio condicional é reconhecida. A instrução-alvo é armazenada em um registrador, até que a instrução de desvio seja executada. Seja o desvio tomado ou não, a próxima instrução a ser executada terá sido buscada antecipadamente.

Essa abordagem é adotada no 360/91 da IBM.

Memória de laço de repetição

Essa técnica usa uma pequena memória de alta velocidade, mantida pelo estágio de busca de instrução da *pipeline*, que é usada para manter as n instruções buscadas mais recentemente, em seqüência. Essa memória é denominada memória de laço de repetição (*loop buffer*). Caso o desvio seja tomado, o hardware verifica primeiramente se a instrução-alvo do desvio está armazenada nessa área de memória. Em caso afirmativo, a próxima instrução é buscada nessa área. O uso de memória de laço de repetição apresenta três vantagens:

1. Com o uso de busca antecipada, a memória de laço de repetição conterá certo número de instruções que estão à frente da instrução corrente na seqüência de instruções do programa. Portanto, as instruções seguintes estarão disponíveis sem requerer o tempo usual de acesso à memória.
2. Se ocorrer um desvio para um endereço-alvo localizado apenas algumas posições adiante do endereço da instrução de desvio, esse alvo já estará na memória de laço de repetição. Isso é útil por ser comum a ocorrência de seqüências de comandos IF-THEN e IF-THEN-ELSE.
3. Essa estratégia é particularmente adequada para lidar com laços de repetição ou iterações; daí o nome *memória de laço de repetição*. Se essa memória for suficientemente grande para conter todas as instruções de um laço de repetição, essas instruções terão de ser buscadas da memória apenas uma vez, para a primeira iteração. Nas iterações subseqüentes, todas as instruções necessárias já estarão nessa área de memória.

A memória de laço de repetição é semelhante, em princípio, a uma memória cache dedicada para instruções. A diferença é que a memória de laço de repetição retém apenas algumas instruções seqüenciais e tem tamanho muito menor do que uma cache de instruções, tendo, portanto, menor custo.

A Figura 11.15 mostra um exemplo de memória de laço de repetição. Ao supor que essa memória contém 256 bytes e que é usado endereçamento de byte, os 8 bits menos significativos do endereço de desvio são usados para indexar essa área de memória. Os bits mais significativos restantes são verificados para determinar se o alvo da instrução de desvio está contido na memória de laço de repetição.

Entre as máquinas que adotam essa abordagem estão algumas das máquinas CDC (Star-100, 6600, 7600) e o CRAY-1. O Motorola 68010 dispõe de uma forma especializada de memória de laço de repetição, usada para executar um laço de três instruções envolvendo a instrução DBcc (Decrementar e Desviar de Acordo com a Condição — veja o Exercício 11.6). Essa memória especial tem tamanho de três palavras e o processador executa as instruções armazenadas nessa memória repetidamente, até que a condição de saída do laço de repetição seja satisfeita.

Previsão de desvio

Várias técnicas podem ser usadas para prever se um desvio será tomado ou não. Entre as mais comuns estão as seguintes:

- Prever que o desvio nunca será tomado
- Prever que o desvio sempre será tomado
- Prever se o desvio será tomado ou não conforme o código de operação
- Prever o desvio com base em chaves de desvio tomado e de desvio não tomado
- Prever o desvio com base em uma tabela de histórico de desvios

As três primeiras abordagens anteriores são estáticas: elas não dependem do histórico de execução de instruções até o momento em que ocorre a instrução de desvio condicional. As duas últimas abordagens são dinâmicas: elas dependem do histórico de execução.

As duas primeiras abordagens são as mais simples. A primeira supõe sempre que o desvio não será tomado e continua a buscar instruções na seqüência em que ocorrem no programa. A segunda supõe sempre que o desvio será tomado, buscando sempre as próximas instruções a partir do endereço-alvo do desvio. O Motorola 68020 e o VAX 11/780 adotam a abordagem de previsão em que o desvio nunca será tomado. O VAX 11/780 inclui também uma característica para minimizar o efeito de uma decisão errada. Se a busca da instrução consecutiva à instrução de desvio causar uma falta de página ou uma violação de proteção, o processador interromperá a busca antecipada da instrução, até que tenha certeza de que essa instrução deve ser mesmo buscada.

Análises de comportamento de programas mostram que desvios condicionais são tomados em mais de 50% das vezes (Lilja, 1988). Portanto, se o custo da busca antecipada de instruções for o mesmo em qualquer caminho, o resultado obtido deverá ser melhor se a busca antecipada de instruções for sempre efetuada a partir do endereço-alvo do desvio. Entretanto, em uma máquina que usa paginação, a busca antecipada de instruções, a partir do endereço de desvio, tem maior probabilidade de causar uma falta de página do que a busca de instruções consecutivas à instrução de desvio. Essa perda de desempenho deve ser levada em conta e algum mecanismo deve ser empregado para reduzir essa penalidade.

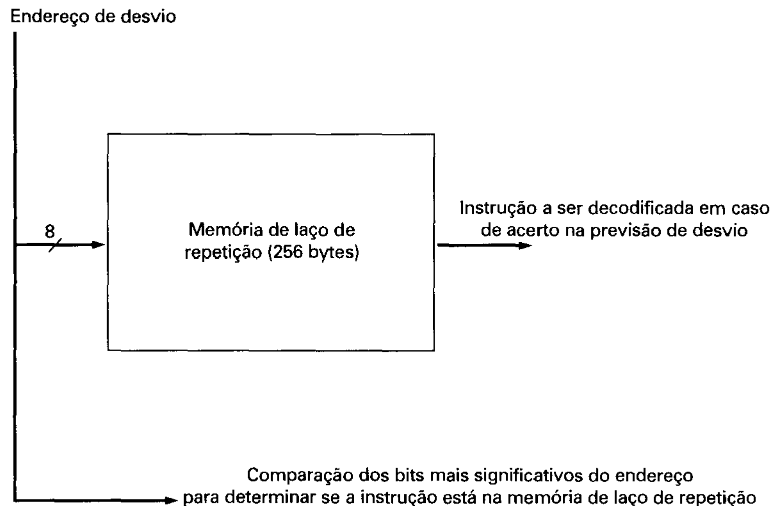


Figura 11.15 Memória de laço de repetição.

A última das abordagens estáveis toma a decisão de previsão de desvio com base no código de operação da instrução de desvio. No caso de determinados códigos de operação, o processador supõe que o desvio sempre é tomado; em outros, que o desvio nunca é tomado. Um estudo feito por Lilja (1988) relata taxas de sucesso superiores a 75% com essa estratégia.

Estratégias dinâmicas de previsão de desvio buscam melhorar a exatidão da previsão mantendo um histórico sobre as instruções de desvio condicional de um programa. Por exemplo, podem ser associados um ou mais bits a cada instrução de desvio condicional, usados para refletir a história recente da execução da instrução. Esses bits são conhecidos como chaves de desvio tomado ou de desvio não tomado e guiam a decisão do processador sobre o caminho a tomar na próxima vez em que a instrução é encontrada. Os bits de histórico tipicamente não são associados à instrução na memória principal. Em vez disso, eles são mantidos em uma memória temporária de alta velocidade. Uma possibilidade é associar esses bits a qualquer instrução de desvio condicional que esteja presente na memória cache. Quando a instrução é substituída na cache, seu histórico é perdido. Outra possibilidade é manter uma pequena tabela de histórico de instruções de desvio executadas mais recentemente, incluindo um ou mais bits em cada entrada. O processador pode endereçar essa tabela de maneira associativa, tal como uma memória cache, ou usar bits de mais baixa ordem do endereço das instruções de desvio.

Se é usado apenas um bit de histórico, a única informação que pode ser registrada é se a última execução da instrução resultou em desvio ou não. Uma desvantagem de usar um único bit ocorre no caso das instruções em que o desvio quase sempre é tomado, tal como em instruções de desvio usadas para implementar laços de repetição. Com apenas um bit de histórico, sempre ocorrerão dois erros de previsão de desvio, cada vez que o laço de repetição for usado: uma vez na entrada do laço e outra na saída.

Se são usados dois bits de histórico, é possível registrar o resultado das duas últimas vezes em que a instrução correspondente foi executada ou registrar, de alguma outra maneira, uma informação de estado da instrução. A Figura 11.16 mostra uma abordagem típica (outras possibilidades serão discutidas no Exercício 11.5). O processo de decisão pode ser represen-

tado por uma máquina de estados finitos com quatro estados. Se as duas últimas execuções de uma dada instrução de desvio tomaram o mesmo caminho, a previsão será tomar esse caminho outra vez. Caso essa previsão não seja confirmada, ela permanecerá a mesma na próxima vez em que a instrução for encontrada. Entretanto, se novamente a previsão não se verificar, a próxima previsão será para selecionar o caminho oposto. Portanto, para mudar a previsão, o algoritmo requer duas previsões erradas consecutivas. Com essa estratégia, se um desvio tomar um caminho não usual apenas uma vez, tal como no caso de um laço de repetição, a previsão será errada apenas uma vez.

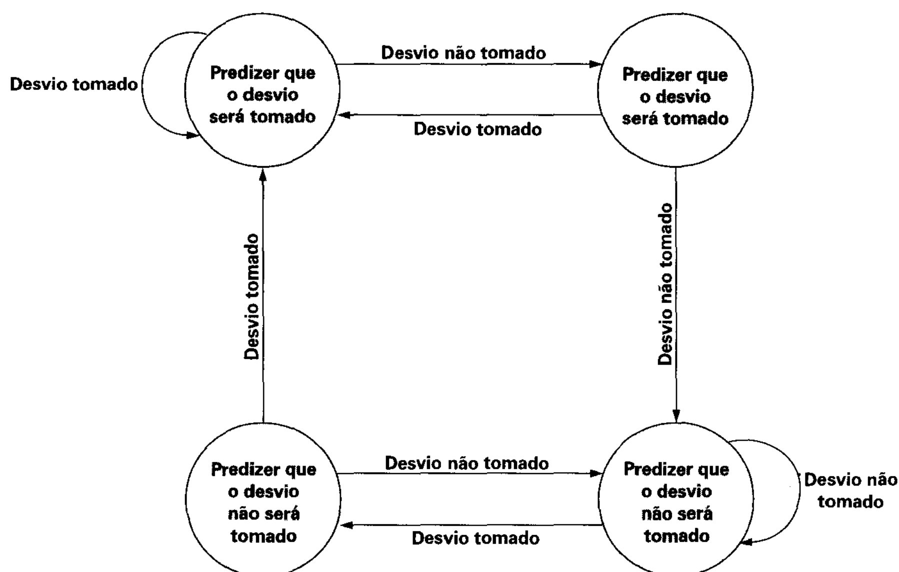
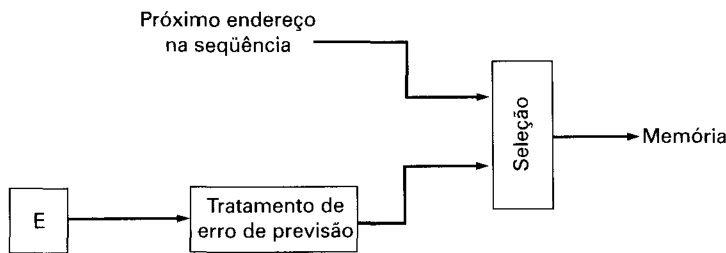


Figura 11.16 Diagrama de transição de estados para previsão de desvio.

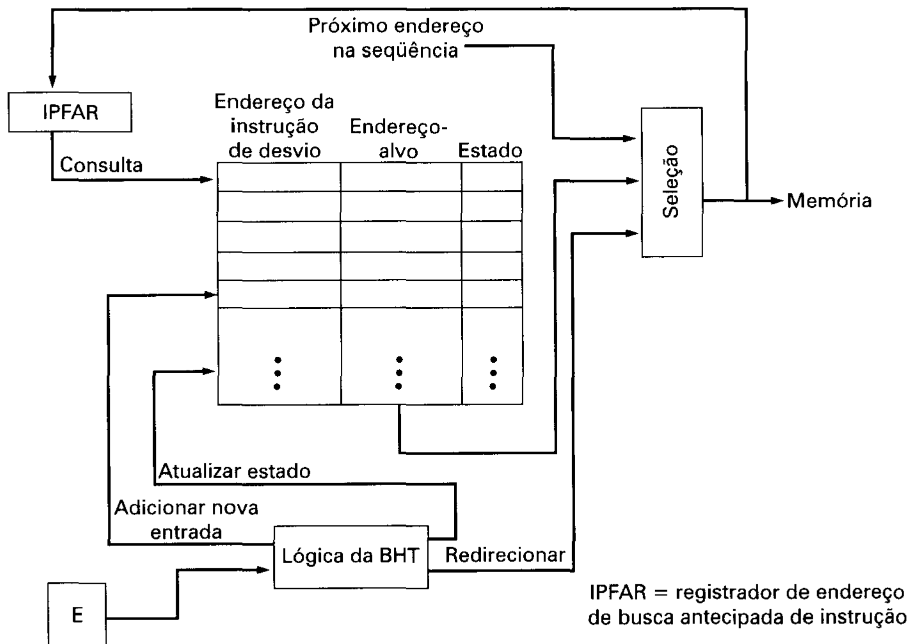
O uso de bits de histórico, tal como descrito anteriormente, tem uma desvantagem: se a decisão é tomar o desvio, a instrução-alvo apenas pode ser buscada quando seu endereço, que é um operando da instrução de desvio, for decodificado. Poderíamos obter maior eficiência se a busca da próxima instrução pudesse ser iniciada assim que a previsão de desvio fosse feita. Para isso, é necessário salvar outras informações em uma estrutura de dados conhecida como tabela de alvos de desvios (*branch target buffer*) ou tabela de histórico de desvios (*branch history table* — BHT).

A tabela de histórico de desvios é uma pequena memória cache associada ao estágio de busca de instrução da *pipeline*. Cada entrada na tabela consiste em três elementos: o endereço de uma instrução de desvio, certo número de bits de histórico, que registram o estado de uso dessa instrução, e alguma informação sobre a instrução-alvo. Na maioria das propostas e das implementações, esse terceiro campo contém o endereço da instrução-alvo. Outra possibilidade é que esse campo contenha a própria instrução-alvo. As vantagens e as desvantagens de uma proposta em relação à outra são claras: a estratégia de armazenar o endereço-alvo resulta em uma tabela menor, mas também em um tempo de busca de instrução maior, em comparação com a estratégia de armazenar a própria instrução-alvo (Reches, 1998).

A Figura 11.17 compara esse esquema com o da previsão em que o desvio nunca será tomado, em que o estágio de busca de instrução sempre obtém a próxima instrução no endereço consecutivo ao da instrução de desvio. O processador usa uma lógica para detectar quando um desvio é tomado e, nesse caso, instrui o estágio de busca para obter a próxima instrução no endereço-alvo (além de, conseqüentemente, esvaziar a *pipeline*). A tabela de histórico de desvios é tratada como uma memória cache. Cada busca antecipada de instrução dispara uma consulta a essa tabela. Quando a entrada correspondente à instrução de desvio não é encontrada na tabela, o endereço consecutivo a essa instrução é usado para a busca da próxima instrução. Quando a entrada correspondente à instrução de desvio é encontrada, a previsão é feita com base no estado dessa instrução: o endereço consecutivo ao desvio ou o endereço-alvo do desvio é informado para a lógica de seleção.



(a) Estratégia de previsão de desvio nunca tomado



(b) Estratégia da tabela de histórico de desvios

Figura 11.17 Lidando com desvios.

Quando é executada uma instrução de desvio, o estágio de execução envia um sinal para a lógica da tabela de histórico de desvios, informando o resultado. O estado da instrução é atualizado de modo que refleta se a previsão de desvio foi correta ou incorreta. Se a previsão foi incorreta, a lógica de seleção é redirecionada para o endereço correto da próxima instrução. Caso a instrução de desvio condicional não seja encontrada na tabela, ela será adicionada à tabela e uma das entradas existentes será descartada, usando um dos algoritmos de substituição de blocos da memória cache discutidos no Capítulo 4.

Um exemplo de implementação de tabela de histórico de desvios é encontrado no microprocessador AMD 29000 da Advanced Micro Device.

Atraso de desvio

O desempenho da *pipeline* também pode ser melhorado ao usar uma técnica para reordenar automaticamente as instruções de um programa, de modo que instruções de desvio ocorram mais tarde do que ocorrem de fato na seqüência especificada. Essa intrigante abordagem é examinada no Capítulo 12.

Pipeline do Intel 80486

O Intel 80486 implementa uma *pipeline* de cinco estágios:

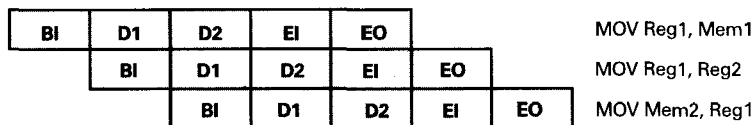
- **Estágio de busca (BI):** as instruções são buscadas na cache ou na memória externa e são colocadas em uma das duas áreas de armazenamento temporário reservadas para busca antecipada de instruções, cada qual com 16 bytes. O objetivo do estágio de busca é preencher essas duas áreas de armazenamento com novas instruções, assim que as instruções anteriores tenham sido consumidas pelo decodificador de instrução. Como as instruções têm tamanho variável (de 1 a 11 bytes, sem contar os prefixos), o estado da busca antecipada de instruções com relação aos demais estágios da *pipeline* varia de instrução para instrução. Em média, são buscadas cerca de cinco instruções a cada carga de 16 bytes (Crawford, 1990). O estágio de busca opera independentemente dos outros estágios, procurando manter cheias as áreas de armazenamento temporário de busca antecipada de instruções.
- **Estágio de decodificação 1 (D1):** toda informação de código de operação e de modo de endereçamento é decodificada no estágio D1. A informação requerida, assim como a informação de tamanho da instrução, ocupa no máximo os 3 primeiros bytes da instrução. Portanto, esses 3 bytes são passados da área de armazenamento temporário de busca antecipada de instruções para o estágio D1. O estágio D1 pode, então, dirigir o estágio D2 para que o restante da instrução (dados imediatos e deslocamento) seja obtido, o que não é requerido na decodificação efetuada em D1.
- **Estágio de decodificação 2 (D2):** o estágio D2 expande cada código de operação em sinais de controle para a ULA. Ele controla também a computação de endereços, no caso de modos de endereçamento mais complexos.
- **Estágio de execução de instrução (EI):** esse estágio inclui a execução de operações pela ULA, o acesso à memória cache e a atualização de registradores.

- **Estágio de escrita de resultado (EO):** esse estágio, se necessário, atualiza registradores e códigos de condição modificados durante o estágio de execução precedente. Caso a instrução corrente atualize a memória, o valor computado é enviado, ao mesmo tempo, para a memória cache e para a área de armazenamento temporário de escrita da interface de barramento.

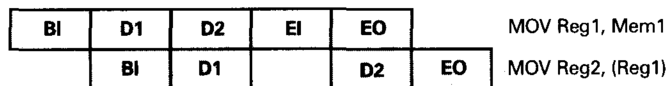
Com o uso de dois estágios de decodificação, a *pipeline* pode sustentar uma taxa de execução de uma instrução por ciclo de relógio. Instruções complexas e desvios condicionais podem diminuir essa taxa.

A Figura 11.18 mostra exemplos de operação da *pipeline*. A parte (a) mostra que não é introduzido qualquer atraso na *pipeline* quando é requerido um acesso à memória. No entanto, como mostra a parte (b), pode haver atraso no caso de valores usados para computar endereços de memória. Ou seja, se um valor é carregado da memória para um registrador e esse registrador é usado como registrador base na próxima instrução, o processador fica parado por um ciclo. Nesse exemplo, o processador efetua um acesso à memória cache, no estágio EI da primeira instrução, e armazena o valor buscado em um registrador, durante o estágio EO. Entretanto, a próxima instrução necessita do valor contido nesse registrador, em seu estágio D2. Quando o estágio D2 se alinha com o estágio EO da instrução anterior, um circuito secundário de caminhos de sinal possibilita ao estágio D2 ter acesso aos mesmos dados usados para escrita pelo estágio EO, economizando um estágio da *pipeline*.

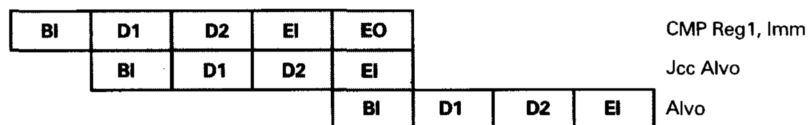
A Figura 11.18c mostra a temporização de uma instrução de desvio, supondo que o desvio é tomado. A instrução de comparação atualiza os códigos de condição no estágio EO e o circuito secundário de caminhos de sinal torna esses códigos disponíveis para o estágio EI da instrução de desvio, ao mesmo tempo. Paralelamente, o processador executa um ciclo de busca especulativo para obter o alvo do desvio, durante o estágio EI da instrução de desvio. Se o processador determinar que a condição de desvio é falsa, ele descartará a busca antecipada e continuará a execução com a próxima instrução da seqüência (já buscada e decodificada).



(a) Sem atraso na *pipeline* devido à carga de dados



(b) Atraso devido à carga de apontador



(c) Temporização da execução de uma instrução de desvio

Figura 11.18 Exemplos de execução de instruções na *pipeline* do 80486.

11.5 O PROCESSADOR PENTIUM II

Uma visão geral da organização do processador Pentium II é apresentada na Figura 4.23. Nesta seção, examinaremos alguns detalhes.

Organização de registradores

O Pentium II inclui os seguintes tipos de registradores (Tabela 11.1):

- **Propósito geral:** existem oito registradores de propósito geral de 32 bits (veja a Figura 11.3c). Esses registradores podem ser usados por todos os tipos de instrução do Pentium II; eles também podem conter operandos para cálculo de endereço. Além disso, alguns desses registradores servem para propósitos especiais. Por exemplo, instruções que manipulam cadeias de caracteres usam os conteúdos dos registradores ECX, ESI e EDI como operandos, sem referenciar explicitamente esses registradores na instrução. Como resultado, diversas instruções podem ser codificadas de modo mais compacto.
- **Segmento:** os seis registradores de segmento de 16 bits cada um contêm seletores de segmento que indexam tabelas de segmentos, como discutido no Capítulo 7. O registrador segmento de código (CS) referencia o segmento que contém a instrução sendo executada. O registrador de segmento de pilha (SS) referencia o segmento que contém a pilha visível para o usuário. Os demais registradores de segmento (DS, ES, FS, GS) possibilitam ao usuário referenciar até quatro segmentos de dados distintos de cada vez.
- **Códigos de condição (*flags*):** o registrador EFLAGS contém códigos de condição e vários bits de controle.
- **Contador de programa:** contém o endereço da instrução corrente.

Há também registradores destinados especificamente para a unidade de ponto flutuante:

- **Numéricos:** cada registrador contém um número de ponto flutuante de precisão estendida de 80 bits. Existem oito registradores que funcionam como uma pilha, havendo disponíveis, no conjunto, instruções, operações para empilhar e desempilhar valores de ponto flutuante.
- **Controle:** o registrador de controle de 16 bits contém bits que controlam a operação da unidade de ponto flutuante, incluindo o controle do tipo de arredondamento, da precisão: simples, dupla ou estendida, e bits para habilitar ou desabilitar várias condições de exceção.
- **Estado:** o registrador de estado de 16 bits contém bits que refletem o estado atual da unidade de ponto flutuante, incluindo um apontador de topo da pilha de 3 bits, códigos de condição, que relatam o resultado da última operação, e indicadores de exceção.
- **Condição de conteúdo (*tag word*):** esse registrador de 16 bits contém 2 bits para cada registrador numérico de ponto flutuante, que indicam a natureza do conteúdo do registrador correspondente. Os quatro possíveis valores são: número válido, zero, especial (NaN, infinito, não-normalizado) e vazio. Essa informação possibilita que programas verifiquem o tipo do conteúdo de um registrador numérico sem ter de efetuar uma decodificação complexa do dado corrente no registrador. Por exemplo, quando é feita uma troca de contexto, o processador não precisa salvar os registradores de ponto flutuante que estejam vazios.

O uso da maioria dos registradores mencionados anteriormente é compreendido facilmente. Descreveremos brevemente alguns desses registradores.

Tabela 11.1 Registradores do processador Pentium II

(a) Unidade inteira

Tipo	Número	Tamanho (bits)	Propósito
Propósito geral	8	32	Registradores de propósito geral para usuários
Segmento	6	16	Contêm seletores de segmento
Códigos de condição	1	32	Bits de estado e de controle
Contador de programa	1	32	Apontador de instruções

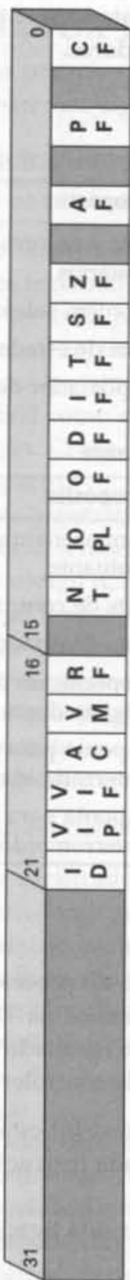
(b) Unidade de ponto flutuante

Tipo	Número	Tamanho (bits)	Propósito
Numérico	8	80	Armazenam um número de ponto flutuante
Controle	1	16	Bits de controle
Estado	1	16	Bits de estado
Bits de condição	1	16	Especificam o conteúdo dos registradores numéricos
Contador de programa	1	48	Aponta para a instrução interrompida pela exceção
Apontador de dados	1	48	Aponta para o operando interrompido pela exceção

Registrador EFLAGS

O registrador EFLAGS (Figura 11.19) indica o estado do processador e ajuda a controlar sua operação. Ele inclui os seis códigos de condição definidos na Tabela 9.8 ('vai-um', paridade, 'vai-um' auxiliar, zero, sinal, *overflow*), que relatam o resultado de uma operação inteira. Além disso, alguns bits do registrador podem ser ditos de controle:

- **Indicador de modo de depuração (TF):** quando esse bit está ligado (valor = 1), é causada uma interrupção depois da execução de cada instrução. Isso é usado para depuração.
- **Habilitação de interrupção (IF):** quando esse bit está ligado (valor = 1), o processador reconhece interrupções externas.
- **Indicador de direção (DF):** determina se as instruções de processamento de cadeias de caracteres incrementam ou decrementam o valor dos registradores de 16 bits SI e DI (para operações de 16 bits) ou dos registradores de 32 bits ESI e EDI (para operações de 32 bits).



ID = Indicador de identificação
VIP = Interrupção virtual pendente
VIF = Habilitação de interrupção virtual
AC = Verificação de alinhamento
VM = Modo virtual 8086
RF = Indicador de reinício
NT = Tarefa aninhada
IOPL = Nível de privilégio de E/S
OF = Bit de *overflow*

DF = Indicador de direção
IF = Habilitação de interrupção
TF = Modo de depuração
SF = Bit de sinal
ZF = Indicador de valor zero
AF = Bit 'vai-um' de auxiliar
PF = Bit de paridade
CF = Bit 'vai-um'

Figura 11.19 Registrador EFLAGS do Pentium II.

- **Indicador de privilégio de E/S (IOPL):** quando esse bit está ligado (valor = 1), o processador gera uma exceção para todo acesso a dispositivos de E/S, durante a operação em modo protegido.
- **Indicador de reinício (RF):** permite ao programador desabilitar exceções de depuração, para que uma instrução possa ser reiniciada depois de uma exceção de depuração sem causar imediatamente uma outra exceção de depuração.

- **Verificação de alinhamento (AC):** quando esse bit é ligado, é feita uma verificação se uma palavra ou palavra dupla é endereçada fora do limite de uma palavra ou palavra dupla.
- **Indicador de identificação (ID):** o valor desse bit apenas pode ser modificado se o processador oferecer suporte para a instrução CPUID. Essa instrução informa o fabricante, a família e o modelo do processador.

Além desses bits, existem 4 bits relacionados ao modo de operação. O bit de tarefa aninhada (NT) indica que a tarefa corrente está aninhada dentro de outra tarefa que opera em modo protegido. O bit de modo virtual (VM) permite ao programador habilitar ou desabilitar o modo virtual 8086, que determina se o processador deve executar como uma máquina 8086. O bit de habilitação de interrupção virtual (VIF) e o bit indicador de interrupção virtual pendente (VIP) são usados em um ambiente multitarefa.

Registradores de controle

O Pentium II emprega quatro registradores de controle de 32 bits (o registrador CR1 não é usado), para controlar vários aspectos de operação de processador (Figura 11.20). O registrador CR0 contém indicadores de controle do sistema, que controlam os modos de operação ou indicam estados que geralmente se aplicam ao processador e não à execução de uma tarefa individual. Esses indicadores são:

- **Habilitação de proteção (PE):** habilita/desabilita o modo de operação protegido.
- **Monitoração de co-processador (MP):** apenas tem interesse quando é executado um programa de máquinas anteriores ao Pentium II; está relacionado com a presença de co-processador aritmético.
- **Emulação (EM):** se esse bit está ligado (valor = 1), isso indica que o processador não possui unidade de ponto flutuante e é gerada uma interrupção em cada tentativa de executar instruções de ponto flutuante.
- **Troca de tarefa (TS):** indica que o processador efetuou uma troca de tarefa.
- **Tipo de extensão (ET):** não é usado no Pentium II; é usado em máquinas anteriores para indicar suporte a instruções do co-processador matemático.
- **Erro numérico (NE):** habilita o mecanismo padrão de reportar erros de ponto flutuante nas linhas do barramento externo.
- **Proteção de escrita (WP):** quando está desligado (valor = 0), as páginas com permissão de usuário apenas para leitura podem ser escritas por um processo supervisor. Essa característica é útil para oferecer suporte à criação de processos em alguns sistemas operacionais.
- **Máscara de alinhamento (AM):** habilita/desabilita a verificação de alinhamento.
- **Desabilita escrita direta (NW):** seleciona o modo de operação da cache de dados. Quando esse bit está ligado (valor = 1), a cache de dados fica impedida de efetuar operações de escrita direta (*write-through*) na cache.
- **Desabilita cache (CD):** habilita/desabilita o mecanismo de abastecimento da cache interna.
- **Paginação (PG):** habilita/desabilita a paginação.

CR4: 31 | 8 PCE | 7 PGE | 6 MCE | 5 PAE | 4 PSE | 3 DE | 2 TSD | 1 PVI | 0 VME

CR3: Base do diretório | PCD | PWT

CR2: Endereço linear de falta de página

CR1

CR0: PG | CD | NW | AM | WP | NE | ET | TS | EM | MP | PE

31 30 29 18 16 5 4 3 2 1 0

PCE = Habilitação de contador de desempenho
PGE = Habilitação de página global
MCE = Habilitação de verificação de máquina
PAE = Extensão de endereço físico
PSE = Extensão de tamanho de página
DE = Extensões de depuração
TSD = Desabilita contador de passos de execução
PVI = Interrupção virtual em modo protegido
VME = Extensões de modo virtual 8086
PCD = Desabilita cache de páginas
PWT = Escrita transparente na cache de páginas

PG = Paginação
CD = Desabilita cache
NW = Desabilita escrita direta na cache
AM = Máscara de alinhamento
WP = Proteção de escrita
NE = Erro numérico
ET = Tipo de extensão
TS = Troca de tarefa
EM = Emulação
MP = Monitoração de co-processador
PE = Habilitação de proteção

Figura 11.20 Registradores de controle do Pentium II.

Quando a paginação é habilitada, os registradores CR2 e CR3 são válidos. O registrador CR2 mantém o endereço linear de 32 bits da última página acessada antes de uma interrupção de falta de página. Os 20 bits mais à esquerda de CR3 correspondem aos 20 bits mais significativos do endereço-base do diretório de páginas; o restante do endereço contém zeros. Dois bits do CR3 são usados para controlar a operação de uma cache externa. O bit de habilitação da cache de páginas (PCD) habilita ou desabilita a operação da cache externa e o bit de escrita transparente na cache de páginas (PWT) controla o modo de escrita na cache externa.

Nove bits de controle adicionais são definidos no CR4:

- **Extensão do modo virtual 8086 (VME):** habilita suporte para o bit indicador de interrupção virtual no modo virtual 8086.
- **Interrupção virtual em modo protegido (PVI):** habilita suporte para o bit indicador de interrupção virtual no modo protegido.
- **Desabilitar contador de passos de execução (TSD):** desabilita a instrução de leitura do contador de passos de execução (RDTSC), que é usado para fins de depuração.
- **Extensões de depuração (DE):** habilita *breakpoints* de E/S; isso permite que o processador interrompa as operações E/S, tanto de leitura quanto de escrita.
- **Extensão de tamanho de página (PSE):** habilita o uso de páginas de 4 Mbytes, no Pentium, ou de páginas de 2 Mbytes, no Pentium Pro e Pentium II.
- **Extensão de endereço físico (PAE):** habilita as linhas de endereço A32 a A35, sempre que um novo modo de endereçamento especial, controlado pela PSE, é habilitado para o Pentium Pro e Pentium II.
- **Habilitação de verificação de máquina (MCE):** habilita interrupção de verificação da máquina, que ocorre quando é detectado erro de paridade durante um ciclo de leitura no barramento ou quando um ciclo de barramento não é completado com sucesso.
- **Habilitação de página global (PGE):** habilita o uso de páginas globais. Quando PGE = 1 e é efetuada uma troca de tarefa, todas as entradas na memória cache da tabela de páginas (*translation lookaside buffer* — TLB) são descarregadas, com exceção daquelas marcadas como globais.
- **Habilitação de contador de desempenho (PCE):** habilita a execução da instrução RDPMC (leitura de contador de desempenho), para qualquer nível de privilégio. Dois contadores de desempenho são usados para medir a duração e o número de ocorrências de um tipo específico de evento.

Registradores MMX

Lembre-se (veja Seção 9.4) de que o Pentium II MMX usa diversos tipos de dados de 64 bits para oferecer suporte de multimídia. As instruções MMX usam campos de endereço de registrador com 3 bits, fornecendo assim suporte para oito registradores MMX. De fato, o processador não inclui registradores MMX específicos. Em vez disso, ele utiliza uma técnica de mapeamento de registradores MMX sobre os registradores de ponto flutuante (Figura 11.21); ou seja, os registradores de ponto flutuante são usados para armazenar valores MMX. Mais especificamente, os 64 bits de mais baixa ordem (mantissa) de cada registrador de ponto flutuante são usados para formar os oito registradores MMX. Assim, a arquitetura existente do Pentium II é facilmente estendida para oferecer suporte para instruções MMX. As características fundamentais do uso dos registradores de ponto flutuante como registradores MMX são:

- Lembre-se de que, nas operações de ponto flutuante, os registradores de ponto flutuante são tratados como uma pilha. Nas operações MMX, esses mesmos registradores são acessados diretamente.
- Na primeira vez em que uma instrução MMX é executada depois de uma operação de ponto flutuante, a palavra de condição de conteúdo da unidade de ponto flutuante é marcada como válida. Isso reflete a mudança do uso dos registradores, que deixam de ser usados sob a forma de pilha para serem usados por meio de endereçamento direto.
- A instrução EMMS (esvazia estado MMX) altera os valores dos bits da palavra de condição de conteúdo da unidade de ponto flutuante, para indicar que todos os registradores estão vazios. É importante que o programador insira essa instrução no final de cada bloco de código MMX, para que as operações de ponto flutuante funcionem adequadamente.
- Quando é escrito um valor em um registrador MMX, é atribuído valor 1 aos bits [79:64] do registrador de ponto flutuante correspondente (bits de sinal e de expoente), indicando que o valor do registrador de ponto flutuante deve ser interpretado como NaN (não um número) ou infinito, quando visto como um valor de ponto flutuante. Isso assegura que um dado MMX nunca será visto como um valor válido de número de ponto flutuante.

Processamento de interrupção

O processamento de interrupções dentro de um processador visa oferecer suporte para o sistema operacional. O uso de interrupções permite que um programa de aplicação seja suspenso, para que uma variedade de condições de interrupção possa ser atendida, sendo a execução do programa retomada mais tarde.

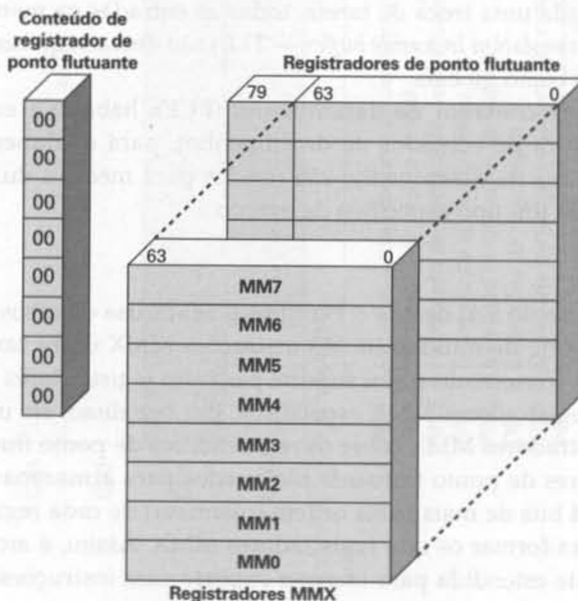


Figura 11.21 Mapeamento de registradores MMX sobre registradores de ponto flutuante.

Interrupções e exceções

Duas classes de eventos fazem com que o Pentium II suspenda o fluxo de execução da instrução corrente e responda ao evento: interrupções e exceções. Em ambos os casos, o processador salva o contexto do processo corrente e transfere o controle para uma rotina predefinida de tratamento da condição ocorrida. Uma *interrupção* é gerada por um sinal de hardware e pode ocorrer aleatoriamente durante a execução de um programa. Uma *exceção* é gerada por software e provocada pela execução de uma instrução. Existem duas fontes de interrupção e duas fontes de exceção, que são:

1. Interrupções
   - **Interrupções mascaráveis:** recebidas no pino INTR do processador. O processador apenas reconhece uma interrupção mascarável se o bit de habilitação de interrupção (IF) tiver valor 1.
   - **Interrupções não-mascaráveis:** recebidas no pino NMI do processador. O reconhecimento dessas interrupções não pode ser desabilitado.
2. Exceções
   - **Exceções detectadas pelo processador:** ocorrem quando o processador detecta um erro ao tentar executar uma instrução.
   - **Exceções programadas:** são exceções geradas por instruções (INTO, INT3, INT e BOUND).

Tabela de vetores de interrupção

O processamento de interrupção no Pentium II usa uma tabela de vetores de interrupção. A cada tipo de interrupção é associado um número, que é usado para indexar a tabela de vetores de interrupção. Essa tabela contém 256 vetores de interrupção de 32 bits, que armazenam o endereço (segmento e endereço relativo) da rotina de tratamento para interrupções daquele tipo.

A Tabela 11.2 mostra os índices de cada tipo de interrupção na tabela de vetores de interrupção; as entradas sombreadas representam interrupções e as não-sombreadas, exceções. A interrupção de hardware NMI é de tipo 2. Interrupções de hardware INTR têm números de 32 a 255; quando é gerada uma interrupção INTR, o sinal de interrupção enviado por meio do barramento deve ser acompanhado do número do vetor de interrupção correspondente. Os demais números de vetores de interrupção são usados para exceções.

Se houver mais de uma interrupção ou exceção pendente, o processador as trata em uma ordem previsível. A localização dos números de vetores na tabela não reflete a prioridade das interrupções. A prioridade de exceções e interrupções é organizada nas cinco classes a seguir, com ordem descendente de prioridade:

- **Classe 1:** parada (*trap*) na instrução anterior (vetor número 1).
- **Classe 2:** interrupções externas (2, 32 a 255).
- **Classe 3:** falhas na busca da próxima instrução (3, 14).
- **Classe 4:** falhas na decodificação da próxima instrução (6, 7).
- **Classe 5:** falhas na execução de uma instrução (0, 4, 5, 8, 10 a 14, 16, 17).

Tratamento de interrupção

Assim como em uma instrução de chamada de procedimento (CALL), a transferência de controle para uma rotina de tratamento de interrupção usa a pilha do sistema para armazenar o estado do processador. Quando uma interrupção ocorre e é reconhecida pelo processador, acontece a seguinte seqüência de eventos:

1. Se a transferência de controle envolver mudança de nível de privilégio, o processador empilha os valores correntes do registrador de segmento de pilha e do registrador de topo de pilha estendido (ESP).
2. O valor corrente do registrador EFLAGS é empilhado.
3. Os bits de interrupção (IF) e de modo de depuração (TF) são desligados (valor = 0). Isso desabilita as interrupções INTR e de depuração geradas depois da execução de cada instrução.
4. O processador empilha o apontador de segmento de código corrente (CS) e o contador de programa corrente (IP ou EIP).
5. Se a interrupção for acompanhada de um código de erro, esse código de erro é colocado na pilha.
6. O conteúdo do vetor de interrupção é obtido e carregado nos registradores CS e IP ou EIP. A execução continua a partir da rotina de tratamento de interrupção.

Para retornar de uma interrupção, a rotina de tratamento de interrupção executa uma instrução IRET. Isso faz com que todos os valores salvos na pilha sejam restabelecidos nos registradores; a execução é retomada no ponto em que ocorreu a interrupção.

Tabela 11.2 Tabela de vetores de interrupção e exceção do Pentium II

Número do vetor	Descrição
0	Erro de divisão; divisão por zero ou *overflow* na divisão
1	Exceção de depuração; inclui várias falhas e indicadores de parada relacionados à depuração
2	Interrupção no pino NMI; sinal no pino NMI
3	Ponto de parada; causado pela instrução INT 3, que é uma instrução de 1 byte, útil para depuração
4	*Overflow* detectado por INTO; ocorre quando o processador executa a instrução INTO e o bit de *overflow* tem valor 1
5	Limite excedido em BOUND; a instrução BOUND compara o conteúdo de um registrador com valores-limite armazenados na memória e gera uma interrupção se o valor do registrador está fora desses limites
6	Código de operação indefinido
7	Dispositivo não disponível; falha na tentativa de usar as instruções ESC ou WAIT devido à ausência de um dispositivo externo
8	Falta dupla; duas interrupções ocorrem durante a execução de uma mesma instrução e elas não podem ser tratadas em seqüência
9	Reservado

Tabela 11.2 Tabela de vetores de interrupção e exceção do Pentium II (*continuação*)

Número do vetor	Descrição
10	Segmento de estado tarefa inválido; o segmento que descreve uma tarefa requerida não está inicializado ou é inválido
11	Segmento não presente; o segmento requerido não está presente
12	Falha de pilha; o limite do segmento de pilha foi excedido ou o segmento de pilha não está presente
13	Proteção geral; uma violação de proteção que não causa outra exceção (por exemplo, escrita em um segmento apenas de leitura)
14	Falta de página
15	Reservado
16	Erro de ponto flutuante; gerado por uma instrução aritmética de ponto flutuante
17	Verificação de alinhamento; acesso a uma palavra armazenada em um endereço de byte ímpar ou a uma palavra dupla armazenada em um endereço que não é múltiplo de 4
18	Verificação de máquina; específica para cada modelo
19-31	Reservado
32-255	Vetores de interrupção de usuário; fornecida quando o sinal INTR está ativado

Não-sombreados: exceções
Sombreados: interrupções

11.6 O PROCESSADOR PowerPC

A Figura 4.25 apresenta uma visão da organização do processador PowerPC. Nessa seção, examinamos alguns detalhes da implementação de 64 bits desse processador.

Organização de registradores

A Figura 11.22 mostra os registradores do PowerPC visíveis para o usuário. A unidade de ponto fixo inclui os seguintes registradores:

- **Propósito geral:** existem 32 registradores de propósito geral, cada um com 64 bits. Esses registradores podem ser usados para carregar, armazenar e manipular operandos de dados e para endereçamento indireto via registrador. O registrador 0 é tratado de modo diferenciado. Nas diversas operações de carga e armazenamento, assim como nas diversas operações de adição, ele é tratado como tendo valor constante, igual a zero, independentemente de seu conteúdo real.
- **Registrador de exceção (XER):** inclui 3 bits que relatam exceções em operações aritméticas de número inteiro. Esse registrador também inclui um campo para um contador de bytes, que é usado como operando em algumas instruções que operam sobre seqüências de caracteres (Figura 11.23a).

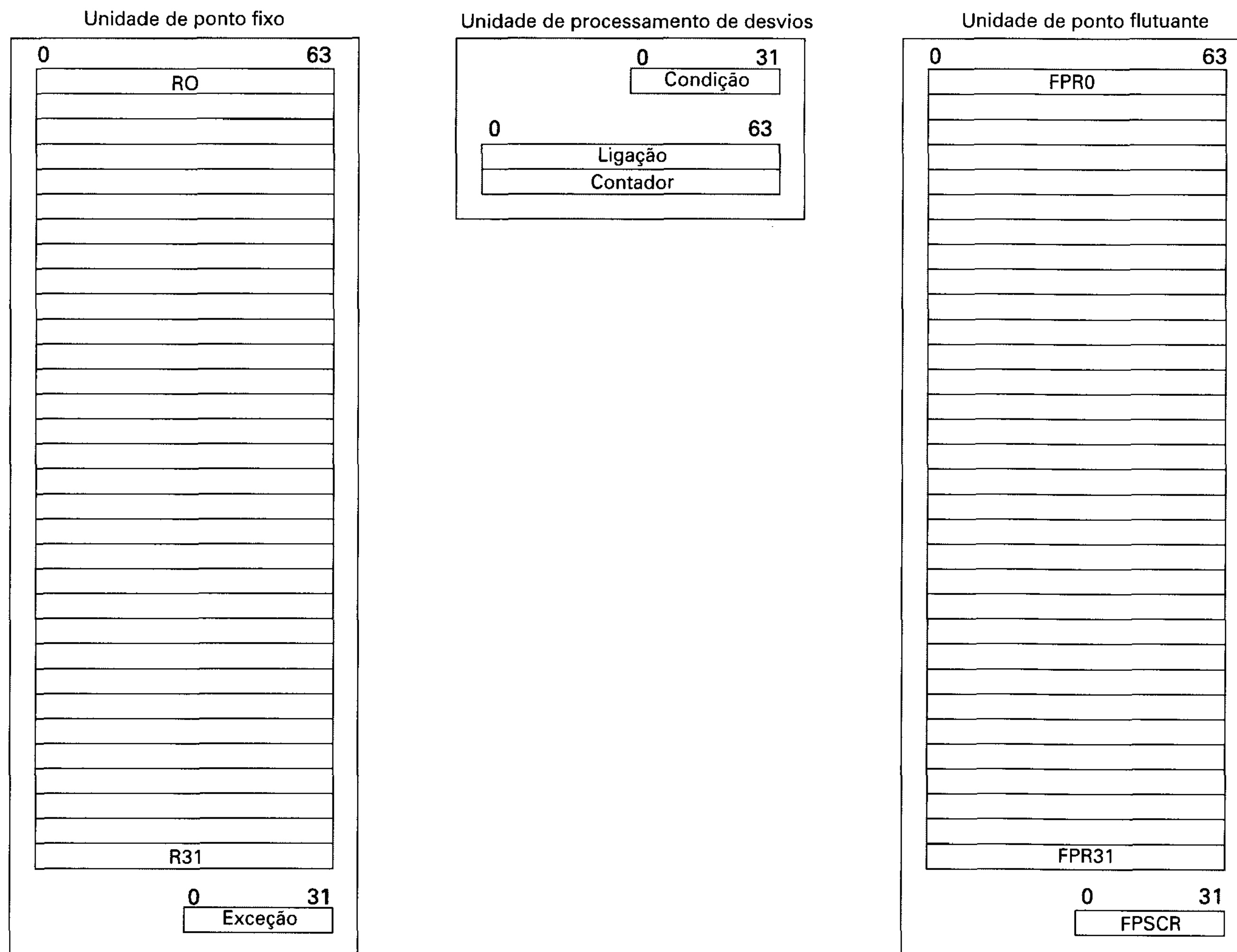
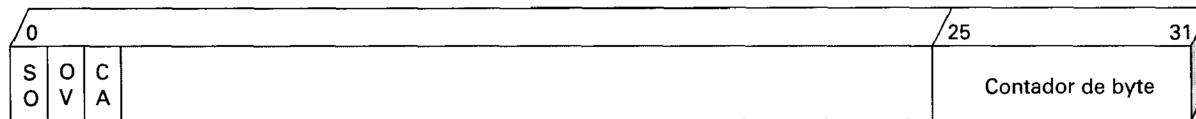


Figura 11.22 Registradores visíveis para o usuário do PowerPC.



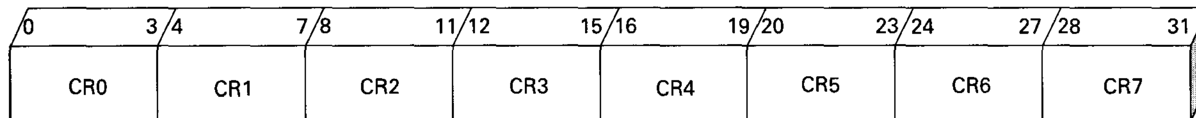
SO = Sumário de *overflow*: esse bit é ligado (valor = 1) para indicar que ocorreu *overflow* durante a execução de uma instrução; o bit é mantido ligado até que seja apagado por software

OV = *Overflow*: esse bit é ligado (valor = 1) para indicar que ocorreu *overflow* durante a execução de uma instrução; seu valor será alterado para 0 pela próxima instrução, se não ocorrer *overflow*

CA = 'Vai-um' (carry): esse bit é ligado (valor = 1) para indicar que ocorreu 'vai-um' para fora do bit de ordem 0 durante a execução de uma instrução

Contador de bytes = especifica o número de bytes a serem transferidos por uma instrução de carga e armazenamento de seqüência de caracteres indexada

(a) Registrador de execução de ponto fixo (XER)



(b) Registrador de condição

Figura 11.23 Formatos de registradores do PowerPC.

A unidade de ponto flutuante contém registradores visíveis para o usuário adicionais:

- **Propósito geral:** existem 32 registradores de propósito geral de 64 bits, usados para todas as operações de ponto flutuante.
- **Registrador de estado e de controle de ponto flutuante (FPSCR):** esse registrador de 32 bits contém bits que controlam a operação da unidade de ponto flutuante e bits que registram o estado resultante de operações de ponto flutuante (Tabela 11.3).

A unidade de processamento de desvios contém os seguintes registradores visíveis para o usuário:

- **Registrador de condição:** consiste de oito campos de código de condição de 4 bits cada, totalizando 32 bits (Figura 11.23b).
- **Registrador de ligação:** o registrador de ligação pode ser usado em uma instrução de desvio condicional com endereçamento indireto ao endereço-alvo. Ele é também usado para implementar a chamada e o retorno de procedimento. Se o bit LK de uma instrução de desvio condicional está ligado, o endereço consecutivo à instrução de desvio é colocado no registrador de ligação e pode ser usado no retorno*.
- **Registrador contador:** o registrador contador pode ser usado para controlar laços de repetição, como foi explicado no Capítulo 9; ele é decrementado cada vez que é testado em uma instrução de desvio condicional. Outro uso desse registrador é no endereçamento indireto do endereço-alvo de uma instrução de desvio.

Os campos do registrador de condição são usados para diversas finalidades. Os primeiros 4 bits (CR0) são atualizados por toda instrução aritmética de número inteiro que tem o bit Rc igual a 1. Como mostra a Tabela 11.4, esse campo indica se o resultado da operação é positivo, negativo ou zero. O quarto bit é uma cópia do bit de sumário de *overflow* do XER. O próximo campo (CR1) é atualizado por toda instrução aritmética de ponto flutuante que tem o bit Rc igual a 1. Nesse caso, os 4 bits são atualizados com o valor dos primeiros 4 bits do FPSCR (Tabela 11.3). Finalmente, qualquer dos oito campos de condições (CR0 a CR7) pode ser usado em uma instrução de comparação; em cada caso, o campo pretendido é especificado na própria instrução. Tanto em uma instrução de comparação de números de ponto fixo quanto em uma comparação de números de ponto flutuante, os primeiros 3 bits do campo de condição especificado registram se o primeiro operando é menor, maior ou igual ao segundo operando. O quarto bit é o bit de sumário de *overflow*, no caso de uma comparação de números de ponto fixo, ou um indicador de resultado não-ordenado, no caso de uma comparação de números de ponto flutuante.

---

* N.R.T.: A idéia do registrador de ligação é proporcionar um retorno de uma sub-rotina mais rápida, pois a CPU não precisará acessar, por exemplo, a pilha para obter o endereço de retorno. É claro que isso se baseia em uma pressuposição de que, normalmente, uma rotina não chama outra, pois nesse caso o registrador de ligação seria sobrescrito.

Tabela 11.3 Registrador de estado e de controle de ponto flutuante do PowerPC

Bit	Definição
0	Sumário de exceção. É ligado (valor = 1) se ocorrer qualquer exceção; permanece ligado até ser apagado por software
1	Sumário de exceção habilitada. É ligado (valor = 1) se ocorrer qualquer exceção habilitada
2	Sumário de exceção de operação inválida. É ligado (valor = 1) se ocorrer uma exceção de operação inválida
3	Exceção de *overflow*. É ligado (valor = 1) se a magnitude do resultado exceder o maior valor que pode ser representado
4	Exceção de *underflow*. É ligado (valor = 1) se o resultado for muito pequeno para ser normalizado
5	Exceção de divisão por zero. É ligado (valor = 1) se o divisor for zero e o dividendo for finito e diferente de zero
6	Exceção de resultado inexato. É ligado (valor = 1) se o resultado arredondado diferir do resultado intermediário ou se ocorrer *overflow* quando a exceção de *overflow* estiver desabilitada
7:12	Exceção de operação inválida. 7: NaN sinalizador; 8: (∞ – ∞); 9: (∞ ÷ ∞); 10: (0 ÷ 0); 11: (∞ × 0); 12: comparação envolvendo NaN
13	Fração arredondada. É ligado se o arredondamento do resultado incrementou a fração
14	Fração inexata. É ligado (valor = 1) se o arredondamento do resultado altera a fração ou se ocorreu *overflow* quando a exceção de *overflow* está desabilitada
15:19	*Flags* de resultado. Código de cinco bits que especifica menor que, maior que, igual, não-ordenado, NaN silencioso, ± ∞, ± normalizado, ± não-normalizado, ±0
20	Reservado
21:23	Exceção de operação inválida. 21: requisição de software; 22: raiz quadrada de número negativo; 23: conversão de número inteiro envolvendo número grande, infinito ou NaN
24	Habilitação de exceção de operação inválida
25	Habilitação de exceção de *overflow*
26	Habilitação de exceção de *underflow*
27	Habilitação de exceção de divisão por zero
28	Habilitação de exceção de resultado inexato
29	Modo não-IEEE
30:31	Controle de arredondamento. Código de dois bits que especifica se o arredondamento é: para o mais próximo, para 0, para + ∞, para –∞

Não-sombreado: bits de estado
Sombreado: bits de controle

Tabela 11.4 Interpretação dos bits no registrador de condição

Posição do bit	CR0 (instrução de número inteiro com Rc = 1)	CR1 (instrução de ponto flutuante com Rc = 1)	CRi (instrução de comparação de ponto fixo)	CRi (instrução de comparação de ponto flutuante)
i	resultado < 0	sumário de *overflow*	op1 < op2	op1 < op2
i + 1	resultado > 0	sumário de exceção desabilitada	op1 > op2	op1 > op2
i + 2	resultado = 0	sumário de exceção de operação inválida	op1 = op2	op1 = op2
i + 3	sumário de *overflow*	exceção de overflow	sumário de *overflow*	Não-ordenado (um operando é um NaN)

Processamento de interrupção

Assim como qualquer processador, o PowerPC oferece um recurso para o processador interromper o programa que está sendo executado, para tratar uma condição de exceção.

Tipos de interrupção

O PowerPC distingue dois tipos de interrupções: as interrupções causadas por alguma condição ou evento do sistema e as causadas pela execução de uma instrução. A Tabela 11.5 apresenta as interrupções reconhecidas pelo PowerPC.

A maioria das interrupções assinaladas na tabela pode ser entendida facilmente, mas algumas requerem maior esclarecimento. A interrupção de reset reinicia o sistema e ocorre quando a energia é ligada e quando o botão de reset da unidade de sistema é pressionado. A interrupção de verificação de máquina lida com certas anomalias, tais como erro de paridade da memória cache e referência a uma posição de memória não existente, e pode fazer com que o sistema entre no que é conhecido como estado de parada para verificação; esse estado suspende a execução do processador e congela os conteúdos de registradores, até que o sistema seja reiniciado. A interrupção de auxílio à unidade de ponto flutuante habilita o processador a invocar rotinas de software para completar operações que não podem ser manipuladas diretamente pela unidade de ponto de flutuante, tais como operações que envolvem números não-normalizados ou códigos de operação de ponto flutuante não implementados.

Tabela 11.5 Tabela de interrupções do PowerPC

Ponto de entrada	Tipo de interrupção	Descrição
00000h	Reservada	
00100h	Reset de sistema	Ativação dos sinais de reset do processador, por hardware ou por software
00200h	Verificação de máquina	Ativação do sinal TEA# para o processador, quando ele está habilitado para reconhecer verificações de máquina
00300h	Acesso a dados	Exemplos: falta de página de dados, violação de direito de acesso em instruções de carga ou de armazenamento
00400h	Acesso a instrução	Falta página de código; tentativa de busca de instrução em um segmento de E/S; violação de direito de acesso
00500h	Externa	Ativação do sinal de entrada de interrupção externa do processador, pela lógica externa quando o reconhecimento de interrupção externa está habilitado
00600h	Alinhamento	Tentativa de acesso à memória malsucedida, devido ao não-alinhamento de operando
00700h	Programa	Interrupção de ponto flutuante; usuário tenta executar uma instrução privilegiada; execução de uma instrução de *trap* em que sua condição é satisfeita; instrução ilegal
00800h	Unidade de ponto flutuante não-disponível	Tentativa de executar instrução de ponto flutuante com a unidade de ponto flutuante desabilitada
00900h	Registrador de decremento	Esgotamento do registrador de decremento quando o reconhecimento de interrupção externa está habilitado
00A00h	Reservada	
00B00h	Reservada	
00C00h	Chamada de sistema	Execução de uma instrução de chamada ao sistema
00D00h	Depuração (*trace*)	Execução de programa passo a passo, para fins de depuração
00E00h	Auxílio à unidade de ponto flutuante	Tentativa de executar uma operação de ponto flutuante pouco freqüente e complexa (por exemplo, uma operação sobre número não-normalizado)
00E10h a 00FFFh	Reserva	
01000h a 02FFFh	Reservada (específica para cada implementação)	

Não-sombreado: interrupções causadas por execução de instrução
Sombreado: interrupções não são causadas por execução de instrução

Registrador de estado da máquina

Para interromper um programa, é fundamental ser capaz de salvar o estado do processador no instante da interrupção, para que possa ser restaurado mais tarde. Isso inclui não apenas salvar o conteúdo de vários registradores mas também várias condições relacionadas à execução. Essas condições são convenientemente armazenadas no registrador MSR (Tabela 11.6). Diversos bits desse registrador também requerem maior esclarecimento.

Quando o bit de modo de privilégio (bit 49) é igual a 1, o processador opera no nível de privilégio de usuário e apenas um subconjunto do conjunto de instruções é disponível. Quando esse bit tem valor 0, o processador opera no nível de privilégio de supervisor. Isso habilita todas as instruções e permite o acesso a certos registradores do sistema (tais como o MSR) que não são acessíveis no nível de privilégio de usuário.

Os valores dos dois bits de exceção de ponto flutuante (bits 52 e 55) definem os tipos de interrupção que a unidade de ponto flutuante pode gerar, tendo a seguinte interpretação:

FE0	FE1	Interrupções que serão reconhecidas
0	0	Nenhuma
0	1	Imprecisão não-recuperável
1	0	Imprecisão recuperável
1	1	Precisão

Quando o bit de depuração passo a passo (bit 53) está ligado, o processador desvia para a rotina de tratamento da interrupção de depuração passo a passo, depois de cada execução de instrução bem-sucedida. Quando o bit de depuração de desvio (bit 54) está ligado, o processador desvia para a rotina de tratamento da interrupção de depuração de desvio, após cada execução de instrução de desvio bem-sucedida, seja o desvio tomado ou não.

Os bits de tradução de endereço de instrução (bit 58) e de tradução de endereço de dados (bit 59) determinam se é usado endereçamento real ou se a unidade de gerenciamento de memória efetua tradução de endereço.

Tratamento de interrupção

Quando uma interrupção ocorre e é reconhecida pelo processador, acontece a seguinte seqüência de eventos:

1. O processador coloca o endereço da próxima instrução a ser executada no registrador SRR0 (*Save/Restore Register 0*). Esse endereço é o endereço da instrução corrente, se a interrupção foi causada por uma falha na tentativa de executar essa instrução, ou, caso contrário, é o endereço da próxima instrução a ser executada.
2. O processador copia a informação de estado da máquina, contida no registrador MSR, para o registrador SRR1. Os bits copiados são os que não estão sombreados na Tabela 11.6. Os bits restantes do registrador SRR1 são carregados com a informação específica para o tipo de interrupção.
3. O registrador MSR é atualizado com um valor definido pelo hardware, específico para o tipo de interrupção. Para todos os tipos de interrupção, a tradução de endereço é desligada e as interrupções externas são desabilitadas.

4. O processador então transfere o controle para a rotina adequada de tratamento de interrupção. Os endereços das rotinas de tratamento de interrupção são armazenados na Tabela de Interrupção (Tabela 11.5). O endereço-base dessa tabela é determinado pelo bit 57 do registrador MSR.

Para retornar de uma interrupção, a rotina de tratamento de interrupção executa uma instrução `rfi` (retorno de interrupção). Isso faz com que os valores dos bits salvos no registrador SRR1 sejam restaurados no registrador MSR. A execução é retomada a partir da instrução cujo endereço está armazenado no registrador SRR0.

Tabela 11.6 Registrador de estado de máquina do PowerPC

Bit	Definição
0	Processador em modo de 32 bits/64 bits
1:44	Reservado
45	Gerenciamento de energia habilitado/desabilitado
46	Dependente de implementação
47	Define se os tratadores de interrupção usam o modo little-endian ou big-endian
48	Interrupção externa habilitada/desabilitada
49	Estado privilegiado/não-privilegiado
50	Unidade de ponto flutuante não-disponível/disponível
51	Interrupções de verificação de máquina habilitada/desabilitada
52	Modo de exceção de ponto flutuante 0
53	Depuração passo a passo habilitada/desabilitada
54	Depuração de desvio habilitada/desabilitada
55	Modo de exceção de ponto flutuante 1
56	Reservado
57	A parte mais significativa do endereço de exceção é 000h/FFFh
58	Tradução de endereço de instrução ligada/desligada
59	Tradução de endereço de dados ligada/desligada
60:61	Reservado
62	Interrupção recuperável/não-recuperável
63	Processador em modo big-endian/little-endian

Não-sombreado: copiado para o SRR1
Sombreado: não copiado para o SRR1

11.7 LEITURA RECOMENDADA

Pipelines de instruções são discutidas detalhadamente em Hennessy (1991) e Hwang (1993). Uma excelente e detalhada discussão sobre as questões envolvidas no projeto do hardware de uma *pipeline* de instruções é apresentada em Sohi (1990). Cragon (1992) apresenta um estudo detalhado sobre previsão de desvio em *pipelines* de instrução. Várias estratégias de previsão de desvio usadas para melhorar o desempenho de *pipelines* são abordadas em Dubey (1991) e Lilja (1988). A dificuldade introduzida na previsão de desvio quando as instruções têm endereço-alvo variável é estudada em Kaeli (1991). A *pipeline* de instruções do Intel 80486 é descrita em Tabak (1991).

O processamento de interrupções do Pentium é descrito em Brey (1997) e o processamento de interrupções do PowerPC é tratado em Shanley (1995a).

11.8 EXERCÍCIOS

11.1 a. Considere um computador com uma palavra de 8 bits. Se a última operação efetuada nesse computador foi uma adição na qual os dois operandos foram 2 e 3, quais seriam os valores dos seguintes bits de condição?

- 'Vai-um'
- Zero
- *Overflow*
- Sinal
- Paridade par
- 'Vai-um' parcial

b. Quais seriam os valores desses bits se os operandos fossem –1 (em complemento de dois) e +1?

11.2 Considere o diagrama de tempo da Figura 11.11. Suponha que existe apenas uma *pipeline* de dois estágios (busca, execução). Redesenhe o diagrama para mostrar quantas unidades de tempo são agora necessárias para quatro instruções.

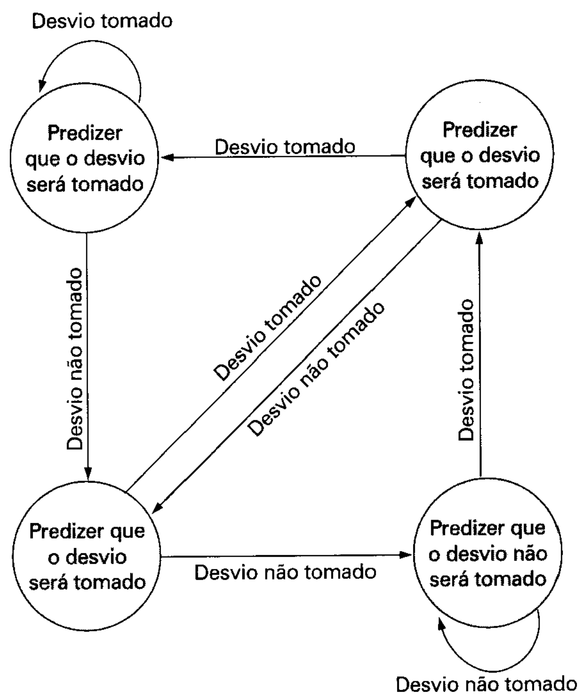
11.3 Considere uma seqüência de instruções de comprimento n fluindo por meio de uma *pipeline* de instruções. Seja p a probabilidade de encontrar uma instrução de desvio condicional ou incondicional e q a probabilidade de que a execução de uma instrução de desvio I ocasione um salto para um endereço não consecutivo. Suponha que um salto desse tipo exija que a *pipeline* seja esvaziada, destruindo todo o processamento de instruções em andamento, quando I emerge do último estágio. Revise as equações 11.1 e 11.2 levando-se em conta essas probabilidades.

11.4 Uma limitação da abordagem de múltiplos fluxos para lidar com desvios em uma *pipeline* é que podem ser encontrados desvios adicionais, antes que o primeiro desvio seja resolvido. Proponha duas outras limitações ou desvantagens.

11.5 Considere os diagramas de transição de estado da Figura 11.24.

a. Descreva o comportamento de cada um dos diagramas.

b. Compare esses diagramas com o diagrama de previsão de desvios da Seção 11.4. Discuta os méritos relativos de cada uma das três abordagens para previsão de desvios.



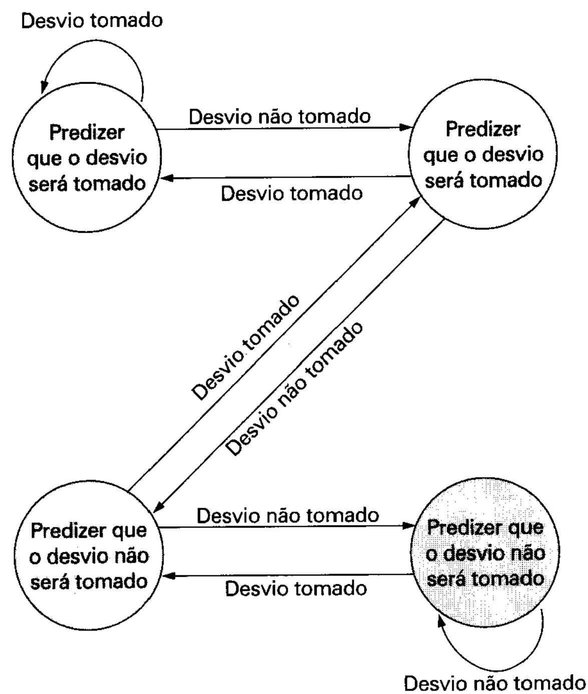


Figura 11.24 Diagramas de transição de estados para o Exercício 11.5.

11.6 As máquinas Motorola 680x0 incluem uma instrução para Decrementar e Desviar de Acordo com a Condição `(DBcc)`, que tem a seguinte forma:

```
DBcc Dn, deslocamento
```

onde `cc` é umas das condições que podem ser testadas, `Dn` é um registrador de propósito geral e deslocamento especifica um endereço-alvo relativo ao endereço corrente. Essa instrução é usada para a implementação de laços de repetição e pode ser definida como a seguir:

```
if (cc = False)
then begin
        Dn := (Dn) - 1;
        if Dn ≠ -1 then PC := (PC) + deslocamento end
else PC := (PC) + 2;
```

Quando a instrução é executada, a condição é testada primeiramente para determinar se a condição de término do laço de repetição é satisfeita. Se isso ocorrer, nenhuma operação será efetuada e a execução continuará na próxima instrução da seqüência. Se a condição é falsa, o registrador de dados específico é decrementado e testado para ver se tem valor menor que zero. Se seu valor for menor que zero, o laço de repetição é terminado e a execução continua na próxima instrução da seqüência. Caso contrário, o programa desvia para a posição especificada. Considere agora o seguinte fragmento de programa em linguagem de montagem:

```
DENOVO    CMPM.L    (A0)1, (A1)1
          DBNE      D1, DENOVO
          NOP
```

Duas seqüências de caracteres endereçadas por A0 e A1 são comparadas, para ver se são iguais; os apontadores para essas seqüências de caracteres são incrementados a cada referência. O registrador D1 contém, inicialmente, o número de palavras longas (4 bytes) a serem comparadas.

a. Os conteúdos iniciais dos registradores são A0 = $000040000, A1 = $000050000 e D1 = $000000FF (o símbolo '$' indica notação hexadecimal). A área de memória entre $40000 e $60000 é carregada com palavras de valor $AAAA. Supondo que seja executado o programa anterior, indique o número de vezes que o laço DBNE é executado e especifique os conteúdos dos três registradores quando a instrução NOP é alcançada.

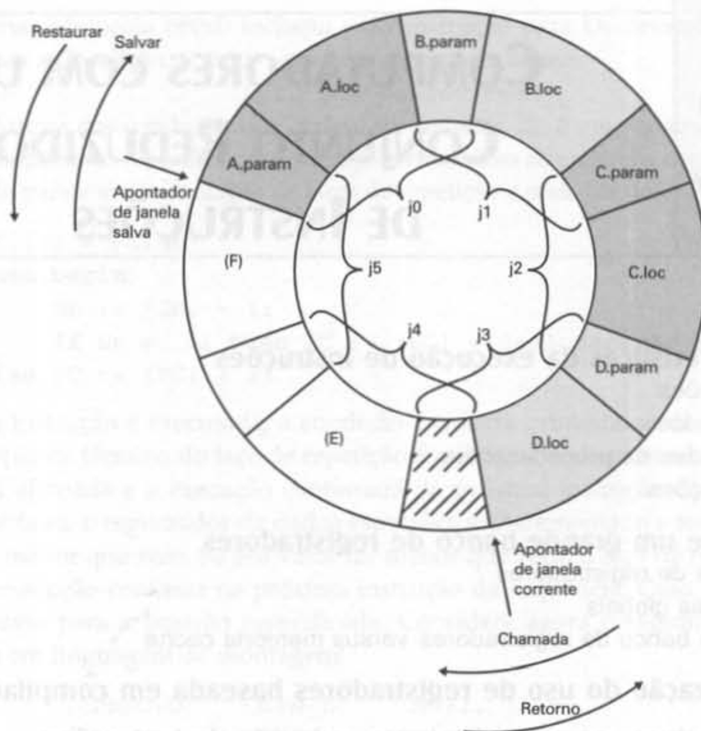
b. Repita o item (a), supondo agora que a área de memória entre $4000 e $4FEE está carregada com $0000 e que a área entre $5000 e $6000 está carregada com $AAA.

11.7 Redesenhe a Figura 11.18c, supondo que o desvio condicional não foi tomado.

capítulo

12

COMPUTADORES COM UM CONJUNTO REDUZIDO DE INSTRUÇÕES



- Estudos sobre o comportamento da execução de programas em linguagens de alto nível orientaram o projeto de um novo tipo de arquitetura para processadores: o computador com um conjunto reduzido de instruções (*reduced instruction set computer* — RISC). Há a predominância de comandos de atribuição, o que sugere que as transferências de dados simples devam ser otimizadas. Além disso, a existência de muitos comandos condicionais (IF) e de laços de repetição sugere que também deve ser otimizado o mecanismo subjacente de controle do seqüenciamento das instruções, de modo que possibilite um uso eficiente de *pipelines*. Estudos sobre padrões de referência a operandos sugerem que é possível obter melhor desempenho, mantendo um número moderado de operandos em registradores.
- Esses estudos motivaram o surgimento de máquinas RISC, com as seguintes características básicas: (1) um conjunto limitado de instruções com formato fixo; (2) um grande número de registradores ou o uso de um compilador que otimize o uso de registradores; e (3) um enfoque na otimização do uso da *pipeline* de instruções.
- O conjunto de instruções simples de uma máquina RISC favorece o uso eficiente de *pipelines* porque o número de operações por instrução é menor e mais previsível. Uma arquitetura com um conjunto reduzido de instruções também favorece o uso da técnica de atraso de instruções de desvio (*delayed branch*), na qual instruções de desvio são trocadas de posição com outras instruções para melhorar a eficiência de uso da *pipeline*.

Desde o desenvolvimento do primeiro computador com programa armazenado na memória, por volta de 1950, houve poucas inovações significativas nas áreas de arquitetura e organização de computadores. Alguns dos maiores avanços desde o nascimento do computador foram:

- **O conceito de família de computadores:** introduzido pela IBM com o Sistema/360, em 1964, e seguido de perto pela DEC, com o PDP-8. O conceito de família de computadores desvincula uma arquitetura de máquina de suas implementações. São fabricados diversos computadores com características de preço e desempenho diferentes, que apresentam, para o usuário, a mesma arquitetura. As diferenças de preço e de desempenho são devidas às diferentes implementações da mesma arquitetura.
- **Unidade de controle microprogramada:** sugerida por Wilkes, em 1951, e introduzida pela IBM na linha S/360, em 1964. A microprogramação facilita a tarefa de projetar e implementar a unidade de controle e oferece suporte para o conceito de família de computadores.
- **Memória cache:** introduzida comercialmente pela primeira vez no IBM S/360 modelo 85, em 1968. A adição desse componente na hierarquia de memória melhorou sensivelmente o desempenho.
- ***Pipeline*:** mecanismo para introduzir paralelismo na natureza essencialmente seqüencial de programas em linguagens de máquina. Alguns exemplos são *pipelines* de instruções e processamento vetorial.
- **Múltiplos processadores:** essa categoria abrange grande número de organizações diferentes, com objetivos distintos.

A essa lista deve ser ainda adicionada uma das inovações mais interessantes e, potencialmente, mais importantes: a arquitetura de computadores com um conjunto reduzido de instruções (RISC). A arquitetura RISC constitui um desvio dramático na história das tendências de arquiteturas de processadores. Uma análise da arquitetura RISC traz à tona muitas das questões importantes relativas à organização e à arquitetura de computadores.

Embora os sistemas RISC tenham sido definidos de várias maneiras por diferentes grupos de projetistas, a maioria dos projetos compartilha os seguintes elementos básicos:

- Um grande número de registradores de propósito geral ou o uso de tecnologias de compilação na otimização do uso de registradores
- Um conjunto de instruções simples e limitado
- Enfoque na otimização da *pipeline* de instruções

A Tabela 12.1 compara diversos sistemas RISC com outros sistemas.

Este capítulo começa com uma breve revisão sobre alguns resultados referentes a conjuntos de instruções e, em seguida, examina cada um dos três tópicos citados anteriormente. Logo depois, são descritos dois dos projetos de máquinas RISC mais bem-documentados.

Tabela 12.1 Características de alguns processadores CISC, RISC e superescalares

	Computador com um conjunto complexo de instruções (CISC)			**Computador com um conjunto reduzido de instruções (RISC)**		**Computador superescalar**		
Característica	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Ano de desenvolvimento	1973	1978	1989	1987	1991	1993	1996	1996
Número de instruções	208	303	235	69	94	225		
Tamanho de uma instrução (bytes)	2-6	2-57	1-11	4	4	4	4	4
Modos de endereçamento	4	22	11	1	1	2	1	1
Número de registradores de propósito geral	16	16	8	40-520	32	32	40-520	32
Tamanho da memória de controle (Kbits)	420	480	246	—	—	—	—	—
Tamanho da cache (Kbytes)	64	64	8	32	128	16-32	32	64

12.1 CARACTERÍSTICAS DA EXECUÇÃO DE INSTRUÇÕES

Uma das formas de evolução mais evidentes associada aos computadores foi a das linguagens de programação. Com a queda do custo do hardware, o custo relativo do software aumentou. Simultaneamente, com a diminuição crônica do número de programadores disponíveis para o desenvolvimento de programas, o custo do software aumentou também em termos absolutos. Assim, o custo do software passou a ser maior, no ciclo de vida de um sistema, do que o do hardware. Além de apresentar um alto custo, o sistema de software é, em geral, pouco confiável: é comum que tanto programas de sistema quanto aplicações continuem a exibir novos erros, mesmo após vários anos em operação.

A resposta dos pesquisadores e da indústria a esses problemas foi desenvolver linguagens de programação de alto nível cada vez mais poderosas e complexas. Linguagens de alto nível possibilitam ao programador expressar algoritmos de maneira mais concisa, abstraindo detalhes de máquina, muitas das quais oferecem suporte à programação estruturada e ao projeto orientado a objetos.

Infelizmente, essa solução fez surgir outro problema, conhecido como *gap semântico*: a enorme distância semântica entre as operações disponíveis em linguagens de alto nível e as operações disponibilizadas pelo hardware de computadores. Os sintomas dessa distância incluem ineficiência na execução de programas, tamanho excessivo dos programas e grande complexidade dos compiladores. Os projetistas responderam a esses problemas buscando desenvolver arquiteturas que diminuíssem a distância entre instruções de linguagens de alto nível e instruções de máquina. Características básicas dessas arquiteturas incluem grandes

conjuntos de instruções, dúzias de modos de endereçamento e implementação de diversos comandos de linguagens de alto nível no hardware da máquina. Um exemplo disso é a instrução de máquina CASE do VAX. Esses complexos conjuntos de instruções tinham os seguintes objetivos:

- Facilitar a tarefa do desenvolvedor de compiladores.
- Melhorar a eficiência da execução de programas, com a implementação de seqüências de operações complexas em microcódigo.
- Oferecer suporte para linguagens de alto nível cada vez mais complexas.

Enquanto isso, durante vários anos foram feitos diversos estudos para determinar as características e os padrões de execução de instruções de máquina geradas por programas em linguagens de alto nível. Os resultados desses estudos inspiraram alguns pesquisadores a buscar uma abordagem diferente: tornar a arquitetura que oferece suporte a linguagens de alto nível mais simples, e não mais complexa.

Para entender essa linha de raciocínio adotada pelos defensores da arquitetura RISC, começamos com uma breve revisão das características da execução de instruções. Os aspectos da computação que devem ser examinados são:

- **Operações realizadas:** essas operações determinam as funções que devem ser realizadas pelo processador e sua interação com a memória.
- **Operandos usados:** os tipos de operandos usados e a freqüência de uso de cada um determinam como a memória deve ser organizada para armazená-los e os modos de endereçamento que devem ser disponíveis para acessá-los.
- **Organização das instruções para execução:** determina o controle e a organização da *pipeline*.

No final desta seção, resumimos os resultados da análise desses aspectos em diversos programas em linguagens de alto nível. Esses resultados são baseados em medições feitas dinamicamente, isto é, coletadas durante a execução de programas, contabilizando o número de ocorrências de alguma característica ou o número de vezes em que se verificou uma determinada propriedade ou condição. Medidas feitas estaticamente, ao contrário, são obtidas examinando exclusivamente o texto do programa fonte. Elas não fornecem informações úteis sobre desempenho, pois não são ponderadas com relação ao número de vezes que cada comando é executado.

Operações

Vários estudos foram realizados para analisar o comportamento de programas de alto nível. A Tabela 4.9, discutida no Capítulo 4, mostra os resultados fundamentais obtidos nesses estudos. Existe grande consenso entre os resultados obtidos com diferentes linguagens e aplicações. O tipo de comando predominante é o comando de atribuição, que sugere que transferências de dados simples são de grande importância. Existe também grande predominância de comandos condicionais e de laços de repetição (IF, LOOP). Esses comandos são implementados em linguagem de máquina que usa algum tipo de comparação e instruções de desvio. Isso sugere que o mecanismo de controle do seqüenciamento das instruções é muito importante.

Os resultados obtidos são instrutivos para o projetista de conjunto de instruções da máquina, indicando os tipos de comandos que ocorrem com maior freqüência e que, portanto,

devem ser implementados de maneira 'ótima'. Entretanto, eles não revelam quais comandos consomem maior parte do tempo de execução de um programa típico. Ou seja, dado um programa compilado para linguagem de máquina, quais comandos da linguagem fonte causam a execução da maioria das instruções de linguagem de máquina?

Para obter essa informação essencial, ou seja, para determinar o número médio de instruções de máquina e referências à memória causadas por tipo de comando de linguagem de alto nível, um conjunto de programas do estudo conduzido por Patterson (1982a), descritos no Apêndice 4A, foi compilado e executado nas máquinas VAX, PDP-11 e Motorola 68000. A segunda e a terceira colunas da Tabela 12.2 mostram a freqüência relativa de ocorrência de várias instruções de linguagens de alto nível em uma variedade de programas; os dados foram obtidos observando as ocorrências durante a execução dos programas e não apenas contando o número de vezes que esses comandos ocorrem no código fonte. Portanto, essas estatísticas constituem freqüências dinâmicas. Para obter os dados apresentados nas colunas 4 e 5 (ponderados para instrução de máquina), cada valor na segunda e na terceira colunas é multiplicado pelo número de instruções de máquina produzido pelo compilador. Esses resultados são então normalizados. Desse modo, as colunas 4 e 5 mostram a freqüência de ocorrência relativa, ponderada pelo número de instruções de máquina por comando da linguagem de alto nível. De maneira análoga, a sexta e a sétima colunas são obtidas multiplicando a freqüência de ocorrência de cada tipo de comando pelo número relativo de referências à memória causado pelo comando. Os dados das colunas 4 a 7 fornecem medidas que correspondem ao tempo gasto efetivamente na execução dos vários tipos de comandos. Os resultados sugerem que a operação de chamada/retorno de procedimento é a que consome mais tempo em programas típicos em linguagens de alto nível.

Tabela 12.2 Freqüência dinâmica relativa ponderada de operações de linguagens de alto nível (Patterson, 1982a)

	Ocorrência dinâmica		**Ponderada por instrução de máquina**		**Ponderada por referência à memória**	
	Pascal	C	Pascal	C	Pascal	C
Comando de atribuição	45	38	13	13	14	15
Comando de repetição	5	3	42	32	33	26
Chamada de procedimento	15	12	31	33	44	45
Comando condicional	29	43	11	21	7	13
Desvio incondicional	—	3	—	—	—	—
Outros	6	1	3	1	2	1

Você deve certificar-se de que compreendeu o significado da Tabela 12.2. Essa tabela indica a importância relativa dos vários tipos de comandos de uma linguagem de alto nível, quando essa linguagem é compilada para uma arquitetura de conjunto de instruções típica. Os resultados seriam possivelmente diferentes para outros tipos de conjuntos de instruções de máquina. Entretanto, esse estudo fornece resultados representativos para arquiteturas modernas de computadores com um conjunto complexo de instruções (*complex instruction set*

computer — CISC). Eles podem, assim, fornecer uma orientação na busca de formas mais eficientes para oferecer suporte a linguagens de alto nível.

Operandos

Um menor número de estudos tem sido realizado para determinar a ocorrência dos diversos tipos de operandos, apesar da importância dessa informação. Existem diversos aspectos significativos.

O estudo de Patterson (1982a), mencionado anteriormente, também procurou determinar a freqüência de ocorrência dinâmica de classes de variáveis (Tabela 12.3). Resultados que foram coerentes entre programas em Pascal e C mostram que a maioria das referências é para variáveis escalares simples, e mais de 80% dessas referências eram para variáveis locais (internas a um procedimento). Além disso, referências a vetores ou registros/estruturas requerem uma referência anterior a seus apontadores ou índices, que são, novamente, variáveis escalares locais. Portanto, existe predominância de referências a variáveis escalares, e essas referências são altamente localizadas.

O estudo de Patterson examinou o comportamento dinâmico de programas em linguagens de alto nível, independentemente da arquitetura subjacente. Como foi discutido anteriormente, para examinar mais profundamente o comportamento de programas é necessário lidar com arquiteturas reais. O estudo apresentado em Lunde (1977) examinou a ocorrência dinâmica de instruções do DEC-10, descobrindo que cada instrução usa, em média, 0,5 operando na memória e 1,4 operando em registradores. Resultados semelhantes são relatados em Huck (1983), para programas C, Pascal e FORTRAN no S/370, PDP-11 e VAX. Embora esses dados dependam fortemente tanto da arquitetura quanto do compilador utilizados, eles mostram efetivamente a freqüência de acesso a operandos.

Esses últimos estudos sugerem a importância de uma arquitetura que facilite o acesso rápido a operandos, pois essa operação é realizada freqüentemente. O estudo de Patterson sugere que um primeiro candidato à otimização é o mecanismo de armazenamento e acesso a variáveis escalares locais.

Tabela 12.3 Porcentagem dinâmica de operandos

	Pascal	C	Média
Constante inteira	16	23	20
Variável escalar	58	53	55
Vetor/registro	26	24	25

Chamadas de procedimentos

Vimos que chamadas e retornos de procedimentos são bastante comuns na execução de programas feitos em linguagens de alto nível. Evidências (Tabela 12.2) sugerem que essas operações são as que consomem mais tempo de execução do código obtido pela compilação de programas. É útil considerar, portanto, mecanismos para implementar essas operações de maneira eficiente. Dois aspectos são significativos: o número de parâmetros e variáveis usadas em um procedimento e o nível de aninhamento de procedimentos.

Um estudo feito por Tanenbaum (1978) mostrou que em 98% das chamadas a procedimentos, durante a execução de um programa, o número de argumentos é inferior a seis e que em 92% dessas chamadas são usadas menos que seis variáveis escalares locais. Resultados semelhantes foram relatados pela equipe de projetistas da arquitetura RISC de Berkeley (Katevenis, 1983), indica a Tabela 12.4. Esses resultados mostram que o número de palavras requeridas por ativação de procedimento não é grande. Estudos relatados anteriormente indicavam que grande parte das referências a operandos são para variáveis escalares locais. Esses estudos mostram que essas referências são, de fato, confinadas a um número relativamente pequeno de variáveis.

O mesmo grupo de Berkeley observou também o padrão de chamadas e retornos de procedimentos em linguagens de alto nível. Eles descobriram que é raro ocorrer uma longa seqüência ininterrupta de chamadas de procedimentos, seguida pela seqüência correspondente de retornos. Ao contrário, eles perceberam que um programa permanece confinado a uma janela bastante estreita de profundidade de chamada de procedimentos. Isso é mostrado na Figura 4.29, discutida no Capítulo 4. Esses resultados reforçam a conclusão de que referências a operandos são altamente localizadas.

Tabela 12.4 Argumentos e variáveis escalares locais de procedimentos

Porcentagem de execução de chamadas de procedimentos com	Compiladores, interpretadores e editores de texto	Pequenos programas não-numéricos
>3 argumentos	0–7%	0–5%
>5 argumentos	0–3%	0%
>8 palavras de argumentos e variáveis escalares locais	1–20%	0–6%
>12 palavras de argumentos e variáveis escalares locais	1–6%	0–3%

Implicações

Diversos grupos de projetistas de computadores observaram os resultados como os que acabamos de relatar e concluíram que a tentativa de projetar uma arquitetura com um conjunto de instruções mais próximo das instruções de linguagens de alto nível não era a estratégia de projeto mais efetiva. Ao contrário, um suporte mais eficaz para linguagens de alto nível poderia ser obtido por meio da otimização do desempenho das características responsáveis por maior consumo de tempo de execução de programas típicos escritos em linguagens de alto nível.

A partir da generalização do trabalho de diversos pesquisadores, podemos perceber que as arquiteturas RISC se caracterizam principalmente por três elementos. Primeiramente, pelo uso de um grande número de registradores ou de técnicas de compilação que visam otimizar a utilização de registradores. Isso visa otimizar referências a operandos. Os estudos discutidos anteriormente mostram que existem diversas referências a operandos em cada instrução de uma linguagem de alto nível e que a proporção de comandos com movimentação de dados (atribuição) é grande. Esse fato, juntamente com a localidade de referências e a predominância

de referências a variáveis escalares, sugere que o desempenho pode ser melhorado com a redução do número de referências à memória, à custa de maior número de referências a registradores. Diante da alta localidade das referências, parece ser útil empregar um conjunto maior de registradores.

Em segundo lugar, é necessária cuidadosa atenção no projeto de *pipelines* de instruções. Em virtude da alta taxa de ocorrência de instruções de desvio condicional e de chamada a procedimento, uma *pipeline* de instruções simples seria ineficiente. Esse fato é comprovado pela grande proporção de instruções buscadas antecipadamente que nunca são executadas.

Finalmente, parece ser mais indicado empregar um conjunto de instruções simplificado (reduzido). Esse ponto não é tão óbvio quanto os outros, mas deverá tornar-se mais claro ao longo da discussão a seguir.

12.2 USO DE UM GRANDE BANCO DE REGISTRADORES

Os resultados resumidos na Seção 12.1 mostram como é importante um acesso rápido a operandos. Vimos que existe grande proporção de comandos de atribuição em programas de linguagens de alto nível, muitos deles da forma simples A ← B. Há também um número significativo de acessos a operandos para cada comando de uma linguagem de alto nível. Juntando esses resultados ao fato de a maioria dos acessos ser de acessos a variáveis escalares locais, fica clara a importância do armazenamento de valores em registradores.

A razão é que, entre os dispositivos de armazenamento disponíveis, os registradores constituem o dispositivo que oferece acesso mais rápido; mais rápido mesmo que a memória principal ou a memória cache. Um banco de registradores é fisicamente pequeno, fica contido na mesma pastilha da ULA e da unidade de controle e usa endereços de tamanho muito menor que endereços da memória cache ou da memória principal. Para tirar proveito disso, é necessário que se adote uma estratégia para garantir que os operandos usados mais freqüentemente sejam mantidos em registradores, minimizando operações de transferência de dados entre os registradores e a memória.

Duas abordagens básicas são possíveis, uma baseada em software e a outra em hardware. A abordagem baseada em software consiste em delegar ao compilador a responsabilidade de otimizar o uso de registradores. O compilador tenta alocar nos registradores as variáveis que serão mais usadas durante um determinado período de tempo. Essa abordagem requer o uso de algoritmos sofisticados de análise de programas. A abordagem baseada em hardware consiste simplesmente em usar um número maior de registradores, de modo que mais variáveis possam ser armazenadas em registradores por um maior período de tempo.

Nesta seção, discutiremos a abordagem de hardware. Essa abordagem foi usada pela primeira vez pelo grupo de projetistas da arquitetura RISC de Berkeley (Patterson, 1982a), no primeiro produto comercial RISC, o processador Pyramid (Ragan-Kelley, 1983), e atualmente é utilizada na popular arquitetura SPARC.

Janelas de registradores

O uso de um grande número de registradores tem como objetivo diminuir a necessidade de acesso à memória. A tarefa do projeto é organizar os registradores de modo que esse objetivo seja atingido.

Como a maioria das referências a operandos é para variáveis escalares locais, a abordagem óbvia é armazenar em registradores os valores dessas variáveis, sendo talvez alguns poucos registradores reservados para variáveis globais. O problema é que a definição de *local* muda com cada chamada e retorno de procedimento, operações que ocorrem com freqüência. Em cada chamada, variáveis locais mantidas em registradores devem ser armazenadas na memória, de maneira que os registradores possam ser reutilizados pelo procedimento chamado. Além disso, devem ser passados parâmetros para o procedimento chamado. No retorno do procedimento, os valores salvos anteriormente devem ser restaurados (carregados de volta nos registradores) e, possivelmente, os resultados devem ser passados do procedimento para o programa que o chamou.

A solução para isso é baseada em dois outros resultados relatados na Seção 12.1. Primeiramente, o fato de um procedimento típico ter um número pequeno de parâmetros e variáveis locais (Tabela 12.4). Em segundo lugar, o fato de a profundidade de ativações de procedimentos variar dentro de uma faixa relativamente estreita (Figura 4.29). Para explorar essas propriedades, são empregados vários conjuntos pequenos de registradores, cada qual alocado para um procedimento diferente. Uma chamada de procedimento faz com que o processador use uma janela de registradores diferente de tamanho fixo, em vez de salvar os valores contidos em registradores na memória. Janelas alocadas para procedimentos adjacentes são sobrepostas, para permitir a passagem de parâmetros.

Esse conceito é mostrado na Figura 12.1. Em qualquer instante de tempo, apenas uma janela de registradores é visível, sendo endereçada como se fosse o único conjunto de registradores disponível (por exemplo, endereços 0 a $N - 1$). A janela é dividida em três áreas de tamanho fixo. Registradores de parâmetros são usados para armazenar os valores dos parâmetros passados do procedimento que efetuou a chamada para o procedimento corrente, assim como os resultados a serem retornados. Registradores locais são usados para variáveis locais, conforme foram designadas pelo compilador. Registradores temporários são usados para trocar parâmetros e resultados com o procedimento de nível inferior seguinte (procedimento chamado pelo procedimento corrente). Os registradores temporários de um nível são, fisicamente, os mesmos que os registradores de parâmetro do próximo nível inferior. Essa sobreposição permite que a passagem de parâmetros seja feita sem a necessidade de movimentação real de dados.

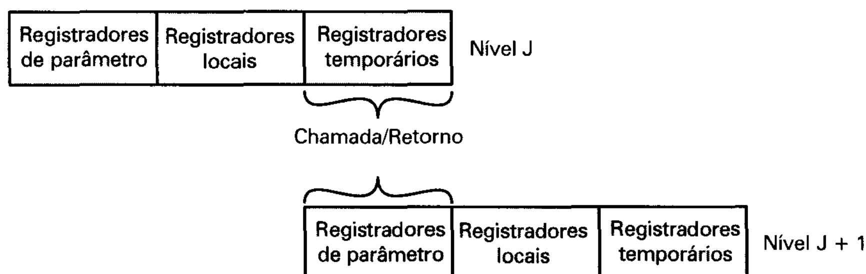


Figura 12.1 Sobreposição de janelas de registradores.

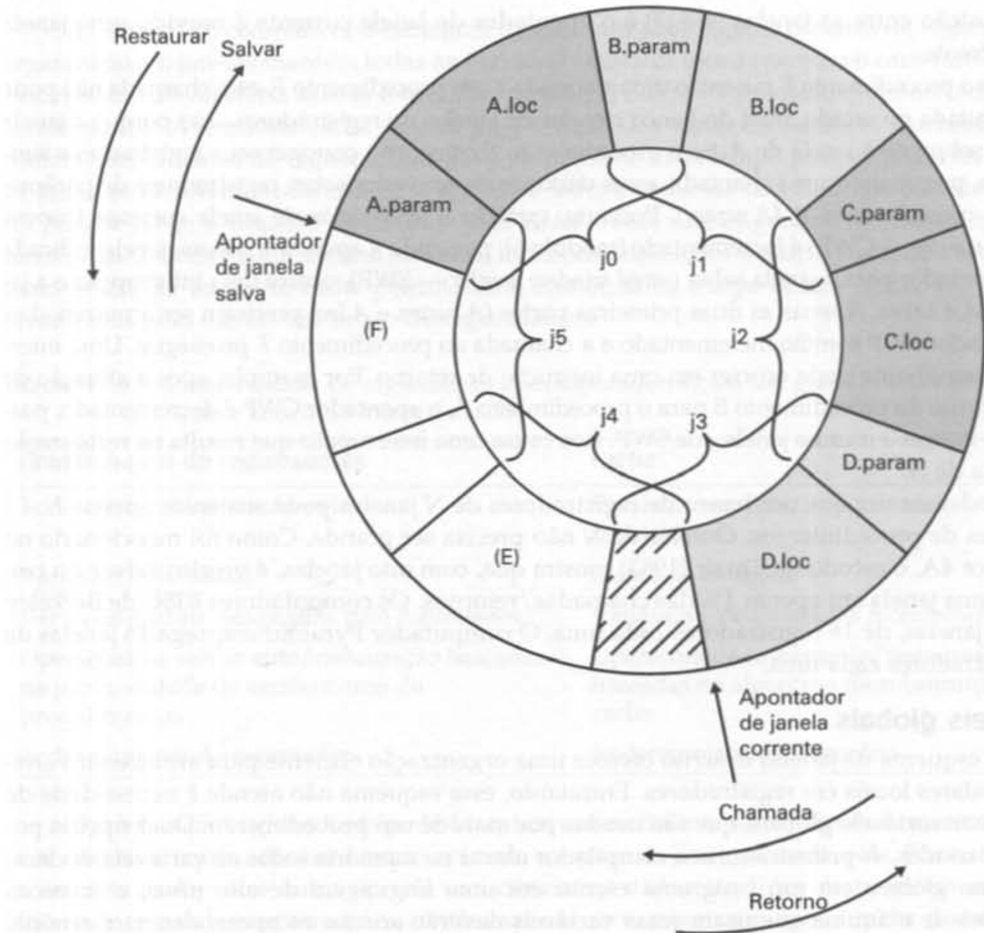


Figura 12.2 Organização circular de janelas de registradores sobrepostas.

Para manipular qualquer possível padrão de chamadas e retornos de procedimentos, o número de janelas de registradores teria de ser ilimitado. Em vez disso, as janelas de registradores podem apenas ser usadas para armazenar dados de algumas poucas ativações de procedimento mais recentes. Dados de ativações mais antigas devem ser salvos na memória e restaurados mais tarde, quando diminuir a profundidade de aninhamento de chamadas. Portanto, o banco de registradores é, de fato, organizado como um arranjo circular de janelas sobrepostas.

Essa organização é mostrada na Figura 12.2, que representa um banco de registradores com seis janelas organizadas de forma circular. A figura mostra a utilização das janelas de registradores em uma chamada de procedimento com nível de aninhamento igual a 4 (*A* chamou *B*; *B* chamou C; C chamou *D*), sendo *D* o procedimento ativo. O apontador para a janela corrente (*current window pointer* — CWP) indica a janela alocada ao procedimento corrente. Uma referência a um registrador em uma instrução de máquina usa esse apontador como base para determinar o registrador físico usado na instrução. O apontador para a janela salva identifica a janela armazenada na memória mais recentemente. Se o procedimento *D* chama um procedimento *E*, os argumentos para *E* são colocados nos registradores temporários de *D* (na

sobreposição entre as janelas j4 e j3) e o apontador de janela corrente é movido uma janela para a frente.

Se o procedimento *E* faz então uma chamada a um procedimento *F*, essa chamada não pode ser executada no estado atual do banco circular de janelas de registradores. Isso porque a janela de *F* se sobrepõe à janela de *A*. Se o procedimento *F* começar a carregar seus registradores temporários, preparando uma chamada, esses dados serão gravados sobre registradores de parâmetros do procedimento *A* (*A.param*). Portanto, quando o apontador de janela corrente (*current window pointer* — CWP) é incrementado (módulo 6), passando a apontar a mesma janela indicada pelo apontador para a janela salva (*saved window pointer* — SWP), ocorre uma interrupção e a janela de *A* é salva. Apenas as duas primeiras partes (*A.param* e *A.loc*) precisam ser armazenadas. O apontador SWP é então incrementado e a chamada ao procedimento *F* prossegue. Uma interrupção semelhante pode ocorrer em uma instrução de retorno. Por exemplo, após a ativação de *F*, no retorno do procedimento *B* para o procedimento *A*, o apontador CWP é decrementado, passando a indicar a mesma janela que SWP. Isso causa uma interrupção que resulta na restauração da janela de *A*.

Podemos ver que um banco de registradores de *N* janelas pode sustentar apenas $N - 1$ ativações de procedimentos. O valor de *N* não precisa ser grande. Como foi mencionado no Apêndice 4A, o estudo de Tmair (1983) mostra que, com oito janelas, é preciso salvar ou restaurar uma janela em apenas 1% das chamadas/retornos. Os computadores RISC de Berkeley usam 8 janelas, de 16 registradores cada uma. O computador Pyramid emprega 16 janelas de 32 registradores cada uma.

Variáveis globais

O esquema de janelas descrito oferece uma organização eficiente para armazenar variáveis escalares locais em registradores. Entretanto, esse esquema não atende à necessidade de armazenar variáveis globais, que são usadas por mais de um procedimento. Duas opções podem ser usadas. A primeira seria o compilador alocar na memória todas as variáveis declaradas como globais em um programa escrito em uma linguagem de alto nível, e todas as instruções de máquina que usam essas variáveis deverão acessar os operandos na memória. Isso é simples, tanto do ponto de vista do hardware como do software (compilador). Entretanto, esse esquema não é eficiente para variáveis globais usadas com muita freqüência.

Uma alternativa é incorporar no processador um conjunto de registradores globais. Haveria um número fixo de registradores desse tipo, disponíveis para todos os procedimentos. Um esquema de numeração unificado poderia ser usado para simplificar o formato das instruções. Por exemplo, referências aos registradores 0 a 7 poderiam mencionar registradores globais únicos e referências aos registradores 8 a 31 corresponderiam a registradores físicos na janela corrente. Isso implica algum aumento de complexidade do hardware, para lidar com essa divisão de endereçamento a registradores. Além disso, o compilador deve decidir que variáveis globais devem ser alocadas nesses registradores.

Grande banco de registradores *versus* memória cache

Um conjunto de registradores organizado em janelas atua como uma área de memória de armazenamento temporário pequena e de acesso rápido, que é usada para manter um subconjunto das variáveis que provavelmente devem ser usadas mais freqüentemente. Sob esse ponto de vista, o banco de registradores atua como uma memória cache. Surge, portanto, a questão de saber se não seria melhor e mais simples usar uma memória cache e o pequeno conjunto tradicional de registradores.

A Tabela 12.5 compara características dessas duas abordagens. O banco de registradores organizado em janelas mantém todas as variáveis escalares locais (exceto no caso raro em que esse número de variáveis exceda o tamanho da janela) dos $N - 1$ procedimentos ativados mais recentemente. A memória cache contém uma seleção das variáveis escalares usadas mais recentemente. O banco de registradores deve economizar mais tempo porque retém os valores de todas as variáveis escalares locais. Por outro lado, a cache faz uso mais eficiente da memória porque reage à situação dinamicamente. Além disso, caches geralmente tratam todas as referências à memória da mesma maneira, incluindo instruções e outros tipos de dados. Portanto, o uso da memória cache permite uma economia de tempo nesses casos, que não são favorecidos pelo uso de um banco de registradores.

Tabela 12.5 Características de organização de computadores com um grande banco de registradores e com uma memória cache

Grande banco de registradores	Cache
Todas as variáveis escalares locais	Variáveis escalares locais usadas recentemente
Variáveis individuais	Blocos de memória
Variáveis globais designadas pelo compilador	Variáveis globais usadas recentemente
Operações de salvamento/restauração baseadas na profundidade de aninhamento de procedimentos	Operações de salvamento/restauração baseadas no algoritmo de substituição de cache
Endereçamento de registrador	Endereçamento de memória

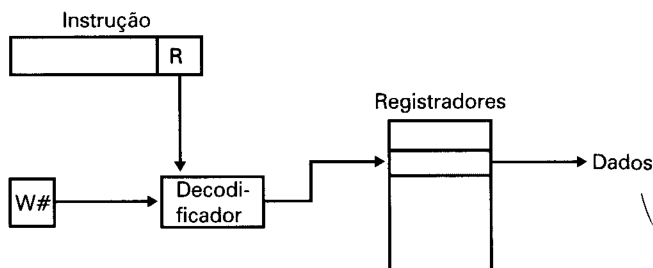
Um banco de registradores pode fazer um uso ineficiente de espaço de armazenamento, pois nem todo o procedimento requererá todo o espaço da janela alocada a ele. Por outro lado, a memória cache sofre outro tipo de ineficiência: os dados são lidos da memória cache em blocos. Enquanto os registradores contêm apenas as variáveis em uso, a cache pode manter um bloco que contenha alguns ou muitos dados que não serão usados.

A cache é capaz de manipular variáveis globais, assim como variáveis locais. Normalmente, existe um grande número de variáveis escalares globais, mas apenas algumas delas são usadas freqüentemente (Katevenis, 1983). O mecanismo de gerenciamento da memória cache é capaz de descobrir dinamicamente as variáveis usadas com freqüência, para mantê-las na cache. Um banco de registradores baseados em janelas, acrescido de registradores globais, também é capaz de reter valores de algumas variáveis escalares globais. Entretanto, é difícil para um compilador determinar as variáveis globais que serão usadas mais freqüentemente.

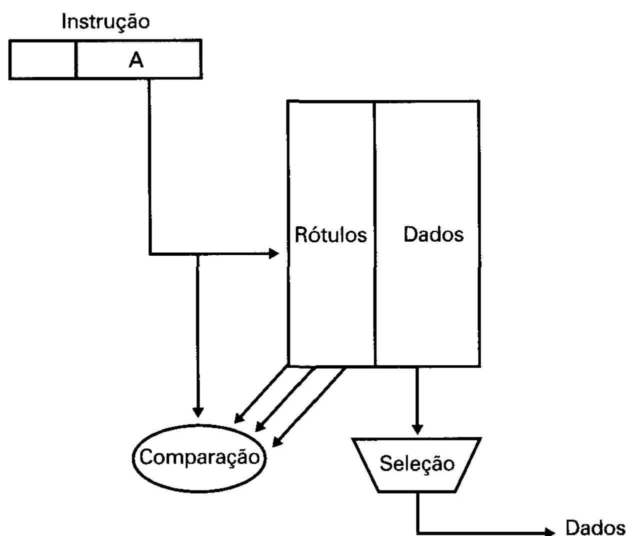
Com o banco de registradores, a transferência de dados entre os registradores e a memória é determinada pela profundidade de aninhamento de procedimentos. Como essa profundidade normalmente varia dentro de uma faixa estreita, o acesso à memória é relativamente raro. A maioria das memórias cache é associativa por conjunto, com um tamanho de conjunto pequeno. Portanto, existe perigo de que outros dados ou instruções substituam variáveis usadas freqüentemente.

Com base na discussão apresentada até aqui, a escolha entre banco de registradores e memória cache não parece óbvia. Entretanto, existe uma característica em que a abordagem de banco de registradores é claramente superior, o que sugere que um sistema baseado em cache seja significativamente mais lento. Essa distinção entre as duas abordagens se revela pela sobrecarga existente no mecanismo de endereçamento de cada abordagem.

A Figura 12.3 mostra essa diferença. Para referenciar uma variável escalar local mantida em um banco de registradores baseado em janelas, um número de registrador 'virtual' e um número de janela são usados. Esses valores são usados por um decodificador relativamente simples para selecionar um registrador físico. Para referenciar uma posição na memória cache, é necessário gerar um endereço de memória completo. A complexidade dessa operação depende do modo de endereçamento. Em uma cache associativa por conjuntos, uma parte do endereço é usada para ler um número de palavras e de rótulos igual ao tamanho do conjunto. Outra parte do endereço é comparada com os rótulos e assim uma das palavras lidas é selecionada. É claro que, mesmo que a cache seja tão rápida quanto o conjunto de registradores, o tempo de acesso será consideravelmente maior. Portanto, do ponto de vista de desempenho, o uso de bancos de registradores baseados em janelas é superior ao uso de cache para armazenar variáveis escalares locais. Um desempenho maior pode ser obtido pela adição de uma cache apenas para instruções.



(a) Banco de registradores baseado em janelas



(b) Cache

Figura 12.3 Referência a um escalar.

12.3 OTIMIZAÇÃO DO USO DE REGISTRADORES BASEADA EM COMPILADORES

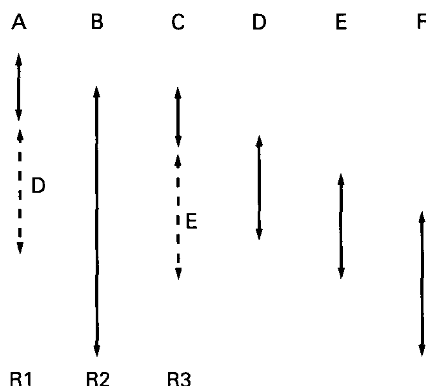
Suponha agora que a máquina RISC alvo dispõe apenas de um pequeno número de registradores (por exemplo, 16 ou 32). Nesse caso, a otimização do uso de registradores é responsabilidade do compilador. Um programa escrito em uma linguagem de alto nível não contém, é claro, referências explícitas a registradores. Ao contrário, os dados são usados no programa de maneira simbólica. O objetivo do compilador é manter em registradores, em vez de na memória, os operandos requeridos no maior número possível de computações, e minimizar operações de transferência de dados entre a memória e os registradores.

Em geral, é adotada a abordagem descrita a seguir. Cada item de dados de um programa, que seja candidato a residir em um registrador, é alocado a um registrador simbólico ou virtual. O compilador então mapeia esse número ilimitado de registradores simbólicos em um número fixo de registradores reais. Registradores simbólicos cujos usos não se sobrepõem podem compartilhar o mesmo registrador real. Caso uma determinada parte de um programa manipule uma quantidade de dados maior que o número de registradores reais, alguns desses dados são armazenados na memória. Instruções são usadas para transferir temporariamente os dados usados nas operações para os registradores.

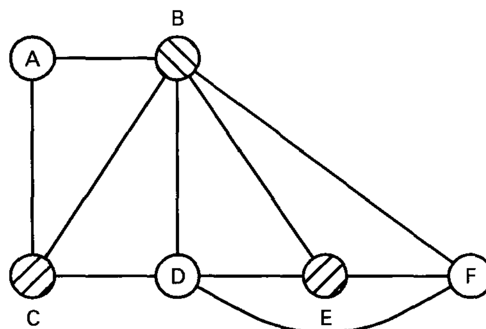
A questão fundamental da tarefa de otimização é decidir quais dados devem ser alocados em registradores em cada ponto da execução do programa. A técnica usada mais comumente em compiladores RISC é conhecida como coloração de grafos, originalmente desenvolvida na disciplina de topologia (Chaitin, 1982; Chow, 1986 e 1990; Coutant, 1986).

O problema da coloração de grafos consiste no seguinte: dado um grafo, constituído de nós e arestas, atribuir cores aos nós, de modo que nós adjacentes tenham cores diferentes, minimizando o número de cores distintas necessárias. Esse problema é adaptado ao problema do compilador para a alocação de dados em registradores, da seguinte maneira. Primeiramente, é feita uma análise do programa, para a construção de um grafo de interferência entre registradores. Os nós do grafo são os registradores simbólicos. Se dois registradores simbólicos estão 'ativos' durante um mesmo trecho de um programa, eles são conectados por uma aresta, que representa uma interferência entre eles. Procura-se então colorir o grafo com até *n* cores, onde *n* é o número de registradores físicos. Nós que compartilhem a mesma cor podem ser alocados ao mesmo registrador. Se esse processo não for totalmente bem-sucedido, os nós que não puderem ser coloridos devem ser alocados na memória, sendo usadas instruções de transferência de dados para liberar espaço no banco de registradores, no instante em que forem requeridos.

A Figura 12.4 apresenta um exemplo simples desse processo. Considere um programa com seis registradores simbólicos, que deve ser compilado de modo que utilize três registradores reais. A Figura 12.4a mostra a seqüência de intervalos em que cada registrador simbólico é usado. A parte b mostra o grafo de interferência entre esses registradores, indicando uma possível coloração do grafo com três cores (hachura e sombreamento são usados no lugar de cores). O registrador simbólico *F* não pode ser colorido e, portanto, devem ser usadas operações de transferência de dados entre registrador e memória para que os valores alocados nesse registrador possam ser acessados.



(a) Seqüência de intervalos de uso ativo de registradores



(b) Grafo de interferência entre registradores

Figura 12.4 Abordagem de coloração de grafos.

Em geral, existe um compromisso entre o uso de um grande número de registradores e de técnicas de otimização do uso de registradores pelo compilador. Um estudo que aborda esse problema é relatado, por exemplo, em Bradlee (1991a), em que uma arquitetura RISC é modelada, com características semelhantes às do Motorola 88000 e do MIPS R2000. O número de registradores varia de 16 a 128, e considera-se tanto o caso em que todos os registradores são de propósito geral como o caso em que os registradores são subdivididos em grupos de registradores de números inteiros e registradores de números de ponto flutuante. O estudo mostra que, mesmo usando uma otimização de registradores simples, existe pouco benefício no uso de mais de 64 registradores. Com técnicas de otimização de registradores razoavelmente mais elaboradas, a melhoria de desempenho já se torna pequena quando são usados mais de 32 registradores. Finalmente, o estudo nota que, se o número de registradores for pequeno (por exemplo, 16), uma máquina com uma organização em que os registradores são compartilhados executará mais rapidamente do que uma máquina em que esses registradores são divididos em dois grupos. Conclusões semelhantes podem ser tiradas de Huguet (1991), que trata de um estudo que enfoca principalmente a otimização do uso de um pequeno número de registradores, e não de uma comparação do uso de um grande conjunto de registradores com a aplicação de técnicas de otimização.

12.4 ARQUITETURA COM UM CONJUNTO REDUZIDO DE INSTRUÇÕES

Nesta seção, abordamos a motivação para uma arquitetura com um conjunto reduzido de instruções e algumas de suas características gerais. Alguns exemplos específicos são examinados mais adiante, neste capítulo. Começamos com uma discussão sobre as motivações para as arquiteturas contemporâneas com um conjunto complexo de instruções (CISC).

Por que CISC?

Notamos anteriormente que existia uma tendência para o uso de conjuntos de instruções cada vez mais ricos, incluindo grande número de instruções e instruções mais complexas. Essa tendência foi motivada por duas razões principais: o desejo de simplificar os compiladores e o desejo de melhorar o desempenho. Essas duas razões, por sua vez, se fundamentavam pela adoção de linguagens de alto nível nas atividades de programação; os projetistas de computadores buscavam projetar máquinas que oferecessem melhor suporte para as linguagens de alto nível.

Não é intenção deste capítulo argumentar que os projetistas de máquinas CISC tenham tomado o caminho errado. De fato, como a tecnologia está permanentemente em evolução e como as arquiteturas existentes variam dentro de um espectro, não podendo ser classificadas em duas categorias distintas — CISC ou RISC —, é pouco provável que algum dia se chegue a uma afirmação conclusiva sobre o tipo de abordagem mais adequada. Portanto, os comentários a seguir visam simplesmente esclarecer alguns dos potenciais problemas da abordagem CISC, fornecendo algum entendimento sobre a motivação dos adeptos da arquitetura RISC.

A princípio, parece óbvio que a abordagem CISC contribuiria para a simplificação de compiladores. Um compilador deve gerar uma seqüência de instruções de máquina para cada comando da linguagem de alto nível. Se existirem instruções de máquina que se assemelhem a comandos de linguagens de alto nível, essa tarefa deve tornar-se mais simples. Entretanto, esse raciocínio vem sendo combatido por pesquisadores adeptos da arquitetura RISC (Hennessy, 1982; Radin, 1983; Patterson, 1982b). Eles descobriram que instruções de máquina complexas freqüentemente são difíceis de ser usadas, porque o compilador tem de encontrar os casos em que essas instruções se adequaram exatamente a uma dada construção da linguagem. Com o uso de um conjunto complexo de instruções, fica muito mais difícil a tarefa de otimizar o código gerado, para minimizar seu tamanho, reduzir o número de instruções executadas e melhorar o desempenho da *pipeline* de instruções. Como prova disso, os estudos citados anteriormente neste capítulo indicam que a maioria das instruções do código gerado por um compilador é relativamente simples.

Outro argumento importante é a expectativa de que uma arquitetura CISC deve produzir programas menores e mais rápidos. Examinamos os dois aspectos dessa afirmativa, ou seja, que os programas serão menores e que sua execução será mais rápida.

Programas menores têm duas vantagens. A primeira é consumir um menor espaço de memória, resultando na economia desse recurso. Como a memória é, hoje em dia, muito barata, essa vantagem potencial deixa de ser tão significativa. A vantagem mais importante de programas menores é, portanto, contribuir para melhorar o desempenho. Isso pode acontecer de duas maneiras. Primeiro, um menor número de instruções significa menor número de by-

tes de instruções a serem buscados. Segundo, em um ambiente de paginação, programas menores ocupam um número menor de páginas, o que reduz a taxa de falta de páginas.

O problema com essa linha de raciocínio é que não se pode ter certeza de que um programa compilado para uma arquitetura CISC será menor que um programa compilado para uma arquitetura RISC correspondente. Em muitos casos, o programa CISC, expresso em linguagem de máquina simbólica, pode ter um número *menor* de instruções, mas o número de bits de memória que ele ocupa pode não ser significativamente *menor*. A Tabela 12.6 mostra resultados de três estudos que comparam tamanhos de programas C compilados para várias máquinas, incluindo a máquina RISC I, que possui uma arquitetura com um conjunto reduzido de instruções. Note que existe pouca ou nenhuma economia com o uso da arquitetura CISC em vez da arquitetura RISC. Também é interessante notar que o VAX, que tem um conjunto de instruções muito mais complexo que o PDP-11, obtém pouca economia sobre esse último. Esses resultados foram confirmados pelos pesquisadores da IBM (Radin, 1983), que descobriram que o IBM 801 (uma máquina RISC) produzia códigos com tamanho igual a 0,9 vez o tamanho do código de um IBM S/370. O estudo utilizou um conjunto de programas escritos em PL/I.

Existem diversas razões para esses resultados um tanto surpreendentes. Vimos que os compiladores para arquiteturas CISC tendem a favorecer instruções mais simples, de modo que a concisão de instruções mais complexas raramente tenha papel significativo. Além disso, como uma arquitetura CISC dispõe de um número maior de instruções, o tamanho do código de operação requerido é maior, resultando em instruções maiores. Finalmente, arquiteturas RISC tendem a enfatizar referências a registradores, em vez de referências à memória, e referências a registradores requerem um menor número de bits. Um exemplo desse efeito será discutido a seguir.

Portanto, a expectativa de que uma arquitetura CISC produziria programas menores, com as correspondentes vantagens, pode não se realizar. A segunda motivação para o uso de conjuntos de instruções cada vez mais complexos era que a execução das instruções seria mais rápida. Parece ter sentido que uma operação de linguagem de alto nível seria executada mais rapidamente se traduzida como uma única instrução de máquina e não como uma série de instruções mais primitivas. No entanto, em razão da tendência de usar as instruções mais simples, isso pode não ser verdade. Para acomodar um conjunto de instruções mais rico, é necessário tornar toda a unidade de controle mais complexa ou usar uma memória de controle de microprograma maior. Qualquer um desses fatores aumenta o tempo de execução de instruções simples.

De fato, alguns pesquisadores descobriram que um aumento na velocidade de execução de funções complexas é obtido não tanto pelo uso de instruções de máquina mais complexas, mas pelo uso de memória de controle de alta velocidade (Radin, 1983). Com efeito, a memória de controle atua como uma cache de instruções. Portanto, o problema do projetista de hardware é determinar quais as sub-rotinas ou funções usadas mais freqüentemente, para alocá-las na memória de controle, implementando-as em microcódigo. Os resultados não têm sido encorajadores. Nos sistemas S/390, instruções como as de divisão de números de ponto flutuante de precisão estendida ou de tradução de códigos de caracteres baseada em tabelas residem em memória de alta velocidade, enquanto a seqüência de microoperações envolvida na preparação de chamadas a procedimentos ou na inicialização de tratamento de interrupção fica na memória principal, mais lenta.

Portanto, não parece muito claro que a tendência no sentido de conjuntos de instruções cada vez mais complexos seja adequada. Isso levou diversos grupos de pesquisadores a buscar o caminho oposto.

Tabela 12.6 Tamanho de código relativo ao RISC I

	(Patterson, 1982a) 11 programas C	(Katevenis, 1983) 12 programas C	(Heath, 1984) 5 programas C
RISC I	1,0	1,0	1,0
VAX-11/780	0,8	0,67	
M68000	0,9		0,9
Z8002	1,2		1,12
PDP-11/70	0,9	0,71	

Características de arquiteturas com conjunto reduzido de instruções

Embora tenham sido adotadas diversas abordagens diferentes para arquiteturas com um conjunto reduzido de instruções, certas características são comuns a todas elas:

- Uma instrução por ciclo
- Operações de registrador para registrador
- Modos de endereçamento simples
- Formatos de instrução simples

Uma breve discussão sobre essas características é apresentada a seguir. Alguns exemplos específicos são examinados no decorrer deste capítulo.

A primeira característica relacionada é que existe **uma instrução por ciclo de máquina**. Um *ciclo de máquina* é definido como o tempo requerido para buscar dois operandos em registradores, executar uma operação da ULA e armazenar o resultado em um registrador. Portanto, instruções de máquinas RISC não devem ser mais complicadas que microinstruções de máquinas CISC (discutidas na Parte 4) e devem executar tão rapidamente quanto essas. Com instruções simples de um ciclo, existe pouca ou nenhuma necessidade de uso de microcódigo; as instruções de máquina podem ser implementadas diretamente pelo hardware. Essas instruções devem executar mais rápido que instruções de máquina semelhantes em outras máquinas, porque não é necessário usar a memória de controle de microprograma durante a execução da instrução.

Uma segunda característica de arquiteturas RISC é que a maioria das operações deve ser de **registrador para registrador**, havendo apenas operações simples de CARGA e ARMAZENAMENTO para acesso à memória. Essa característica de projeto simplifica o conjunto de instruções e, conseqüentemente, a unidade de controle. Por exemplo, um conjunto de instruções RISC pode incluir apenas uma ou duas instruções ADD (por exemplo, adicionar número inteiro e adicionar com 'vai-um'); o VAX tem 25 instruções de adição diferentes. Outro benefício é que essa arquitetura encoraja a otimização do uso de registradores, de maneira que operandos usados freqüentemente sejam mantidos em áreas de armazenamento de alta velocidade.

Essa ênfase em operações de registrador para registrador é particular de projetos RISC. Outras máquinas contemporâneas oferecem instruções desse tipo, mas também incluem operações de memória para memória e operações que combinam registrador e memória. Nos anos 70, antes do aparecimento da arquitetura RISC, foram feitas algumas tentativas de comparar essas abordagens. A Figura 12.5a mostra a estratégia de comparação adotada. Foram avaliadas arquiteturas hipotéticas, em relação ao tamanho de programas e ao tráfego de dados entre os registradores e a memória, em número de bits. Resultados como esses levaram um pesquisador a sugerir que futuras arquiteturas não deveriam conter nenhum registrador (Myers, 1978). Imagine o que ele teria pensado, naquela época, sobre a máquina RISC comercializada pela Pyramid, que contém nada menos que 528 registradores!

O que esses estudos não foram capazes de perceber é que um pequeno número de variáveis escalares locais era usado freqüentemente e que, com um grande banco de registradores ou um compilador otimizador, a maioria dos operandos poderia ser mantida em registradores por longos períodos de tempo. Portanto, a Figura 12.5b parece ser uma comparação mais adequada.

Uma terceira característica das arquiteturas RISC é o uso de **modos de endereçamento simples.** Quase todas as instruções RISC usam endereçamento simples de registrador. Diversos modos de endereçamento adicionais, como endereçamento por deslocamento ou relativo ao PC, podem ser incluídos. Outros modos de endereçamento mais complexos podem ser implementados por software, a partir dos modos de endereçamento mais simples. Novamente, essa característica de projeto simplifica o conjunto de instruções e a unidade de controle.

Uma última característica é o uso de **formatos de instrução simples.** Geralmente, é usado apenas um formato de instrução ou um pequeno número de formatos diferentes. O tamanho da instrução é fixo e alinhado em limites de palavras. As posições dos campos na instrução também são fixas, especialmente a do código de operação. Essa característica de projeto traz diversos benefícios. Com campos de posição fixa, a decodificação do código de operação e o acesso a operandos em registradores podem ser feitos simultaneamente. Formatos simples simplificam também a unidade de controle. A busca de instruções é otimizada pela busca em unidades de palavra. O uso de um alinhamento em limites de palavras faz com que uma instrução não cruze os limites de uma página.

Analisando essas características em conjunto, pode-se determinar os potenciais benefícios da abordagem RISC. Esses benefícios se dividem em duas categorias principais: aqueles relacionados ao desempenho e aqueles relacionados à implementação VLSI.

Com relação ao desempenho, pode ser apresentada certa quantidade de 'evidências circunstanciais'. A primeira é que compiladores otimizadores podem ser desenvolvidos. O uso de instruções primitivas oferece maior oportunidade para mover funções para fora de laços de repetição, reordenando o código para obter maior eficiência, maximizar a utilização de registradores etc. É possível até mesmo calcular parcialmente o resultado de instruções complexas em tempo de compilação. Por exemplo, considere a instrução MVC do S/390, que move uma seqüência de caracteres de um local para outro. Cada vez que essa instrução é executada, o número de movimentações de dados requeridas depende do tamanho da seqüência, de características de alinhamento e se os locais de origem e de destino se sobrepõem. Na maioria dos casos, esses parâmetros são conhecidos em tempo de compilação. Portanto, o compilador pode produzir uma seqüência otimizada de instruções primitivas para desempenhar essa função.

8	16	16	16
Add	B	C	A

Memória para memória
I = 56, D = 96, M = 152

8	4	16	
Load	rB	B	
Load	rC	C	
Add	rA	rB	rC
Store	rA	A	

Registrador para registrador

(a) A ← B + C

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

Memória para memória
I = 168, D = 288, M = 456

8	4	4	4
Add	rA	rB	rC
Add	rB	rA	rC
Sub	rD	rD	rB

Registrador para registrador
I = 104, D = 96, M = 200

(b) A ← B + C; B ← A + C; D ← D − B

I = Tamanho das instruções executadas
D = Tamanho dos dados usados
M = I + D = Tráfego total na memória

Figura 12.5 Duas comparações das abordagens registrador para registrador e memória para memória.

Um segundo ponto, já notado, é que a maioria das instruções geradas por um compilador é relativamente simples. Parece razoável que uma unidade de controle construída especificamente para essas instruções, e que use pouco ou nenhum microcódigo, possa executar instruções mais rapidamente do que uma unidade CISC comparável.

Um terceiro ponto é relacionado ao uso da *pipeline* de instruções. Pesquisadores de arquiteturas RISC argumentam que a técnica de *pipeline* de instruções pode ser aplicada muito mais efetivamente quando existe um conjunto reduzido de instruções. Esse ponto será examinado mais detalhadamente logo a seguir.

Um último ponto que é, de certo modo, menos significativo é que programas em arquiteturas RISC são mais suscetíveis a interrupções, porque a existência de interrupções pendentes é verificada entre operações um tanto elementares. Arquiteturas com instruções complexas restringem a detecção de interrupções a limites de instrução ou devem definir pontos passíveis de interrupção e implementar mecanismos para reiniciar uma instrução.

Tabela 12.7 Esforço de projeto e leiaute de componentes para alguns microprocessadores

CPU	Transistores (milhares)	Projeto (pessoas/mês)	Leiaute de componentes (pessoas/mês)
RISC I	44	15	12
RISC II	41	18	12
M68000	68	100	70
Z8000	18	60	70
Intel iAPx-432	110	170	90

A suposição de que máquinas com um conjunto reduzido de instruções apresentam melhor desempenho está longe de ter sido provada. Embora diversos estudos tenham sido realizados, eles não abordam máquinas de tecnologia e potência comparáveis. Além disso, a maioria dos estudos não procura separar os efeitos de um conjunto reduzido de instruções dos efeitos do uso de um grande banco de registradores. Entretanto, a 'evidência circunstancial' é sugestiva.

A segunda área de benefícios potenciais da arquitetura RISC diz respeito à implementação VLSI. Quando a tecnologia VLSI é usada, o projeto e a implementação do processador são fundamentalmente alterados. Processadores tradicionais, como o IBM S/390 e o VAX, consistem em uma ou mais placas de circuito impresso, com pastilhas padrão SSI ou MSI. Com o advento das tecnologias LSI e VLSI, uma única pastilha pode conter um processador completo. Com processadores em uma única pastilha, existem duas motivações para se adotar a estratégia RISC. A primeira é a questão do desempenho. Atrasos em virtude da transmissão de sinais internos à pastilha têm duração mais curta que atrasos de sinais entre pastilhas distintas. Portanto, tem sentido dedicar o escasso recurso, de circuitos internos à pastilha, a atividades que ocorram freqüentemente. Como vimos anteriormente, as atividades mais freqüentes são, de fato, instruções simples e de acesso a variáveis escalares locais. Os processadores RISC de Berkeley foram projetados tendo em mente esse fato. Enquanto um microprocessador típico que ocupa uma única pastilha dedica cerca de metade da sua área para o armazenamento de microcódigo de controle, a pastilha RISC I dedica apenas cerca de 6% de sua área à unidade de controle (Sherburne, 1984).

Outra questão relacionada ao emprego de tecnologia VLSI é o tempo de projeto e implementação. É difícil desenvolver um processador VLSI. Em vez de se basear em circuitos SSI ou MSI já disponíveis, o projetista tem de desenvolver todo o projeto do circuito, a elaboração da disposição dos componentes internos (leiaute) e fazer a modelagem em nível de dispositivo. Com uma arquitetura com um conjunto reduzido de instruções, esse processo fica muito mais fácil, como mostra a Tabela 12.7 (Fitzpatrick, 1981). Se, além disso, o desempenho da pastilha RISC for equivalente ao de microprocessadores CISC comparáveis, as vantagens da abordagem RISC tornam-se evidentes.

Características CISC *versus* RISC

Depois do entusiasmo inicial por máquinas RISC, reconheceu-se que: (1) projetos RISC podem se beneficiar com a inclusão de algumas características CISC e (2) projetos CISC podem se beneficiar com a inclusão de algumas características RISC. O resultado é que projetos RISC mais recentes, notadamente o PowerPC, não constituem mais projetos RISC 'puros', assim como projetos CISC mais recentes, notadamente o Pentium II, incorporam algumas características RISC.

Uma comparação interessante, apresentada em Mashey (1995), permite melhor entendimento sobre o assunto. A Tabela 12.8 relaciona diversos processadores, comparando-os em relação a várias características. Para fins dessa comparação, as seguintes características são consideradas típicas de um projeto RISC:

1. Tamanho de instrução único.
2. Tamanho típico de instrução de 4 bytes.
3. Pequeno número de modos de endereçamento de dados, usualmente menor que 5. Esse parâmetro é difícil de definir. Nessa tabela, não são contabilizados os modos de endereçamento de registrador ou literal e diferentes formatos, com deslocamentos de tamanhos diferentes, são contabilizados separadamente.

Tabela 12.8 Características de alguns processadores

Processador	Número de tamanhos de instruções	Tamanho máximo de instrução em bytes	Número de modos de endereçamento	Endereçamento indireto	Carga/armazenamento combinado com aritmética	Número máximo de operandos na memória	Possibilidade de endereçamento não-alinhado	Número máximo de usos da MMU	Número de bits para especificar registrador de número inteiro	Número de bits para especificar registrador de ponto flutuante
AMD29000	1	4	1	não	não	1	não	1	8	3[a]
MIPS R2000	1	4	1	não	não	1	não	1	5	4
SPARC	1	4	2	não	não	1	não	1	5	4
MC88000	1	4	3	não	não	1	não	1	5	4
HP PA	1	4	10[a]	não	não	1	não	1	5	4
IBM RT/PC	2[a]	4	1	não	não	1	não	1	4[a]	3[a]
IBM RS/6000	1	4	4	não	não	1	sim	1	5	5
Intel i860	1	4	4	não	não	1	não	1	5	4
IBM 3090	4	8	2[b]	não[b]	sim	2	sim	4	4	2
Intel 80486	12	12	15	não[b]	sim	2	sim	4	3	3
NSC 32016	21	21	23	sim	sim	2	sim	4	3	3
MC68040	11	22	44	sim	sim	2	sim	8	4	3
VAX	56	56	22	sim	sim	6	sim	24	4	0
Clipper	4[a]	8[a]	9[a]	não	não	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	não	não	1	sim[a]	—	5	3[a]

a RISC que não possui essa característica.
b CISC que não possui essa característica.

4. Nenhum endereçamento indireto que faça acesso à memória para obter o endereço de um operando.
5. Nenhuma operação que combine carga ou armazenamento de dados com uma operação aritmética (por exemplo, adicionar valor obtido na memória ou adicionar e armazenar o resultado na memória).
6. Não mais que um operando endereçado na memória por instrução.
7. Inexistência de suporte a alinhamento arbitrário de dados para operações de carga ou armazenamento.
8. Máximo número de usos da unidade de gerenciamento de memória (*memory management unit* — MMU) para endereçamento a dados de uma instrução.
9. Uso de cinco ou mais bits para especificar um registrador de número inteiro. Isso significa que pelo menos 32 registradores de número inteiro distintos podem ser usados a cada vez.
10. Uso de quatro ou mais bits para especificar um registrador de ponto flutuante. Isso significa que pelo menos 16 registradores de ponto flutuante distintos podem ser usados a cada vez.

Os itens 1 a 3 são uma indicação da complexidade de decodificação de instruções. Os itens 4 a 8 sugerem a facilidade ou dificuldade do uso de *pipeline*, especialmente na presença de requisições de acesso a memória virtual. Os itens 9 e 10 são relacionados à capacidade de aproveitar efetivamente o trabalho realizado por compiladores.

Nessa tabela, os primeiros oito processadores são claramente arquiteturas RISC, os cinco processadores seguintes são nitidamente CISC e os dois últimos são processadores em geral classificados como RISC, mas que, de fato, incluem muitas características CISC.

12.5 *PIPELINE* DE INSTRUÇÕES RISC

Pipeline com instruções regulares

Como vimos na Seção 11.4, o uso de *pipelines* de instruções visa melhorar o desempenho. Considere isso no contexto de uma arquitetura RISC. A maioria das instruções é de registrador para registrador, e um ciclo de instrução apresenta as duas fases seguintes:

- *I*: Busca de instrução.
- *E*: Execução. Realiza uma operação da ULA, com entrada e saída em registradores.

A execução de operações de carga e armazenamento requer três fases:

- *I*: Busca de instrução.
- *E*: Execução. Calcula o endereço de memória.
- *D*: Memória. Operação de registrador para memória ou de memória para registrador.

A Figura 12.6a representa o diagrama de tempo da execução de uma seqüência de instruções sem o uso de *pipeline*. Claramente, esse processo apresenta certo desperdício. O desempenho pode ser melhorado substancialmente mesmo com o uso de uma *pipeline* muito simples. A Figura 12.6b mostra um esquema que usa uma *pipeline* de dois caminhos, na qual são executadas simultaneamente as fases *I* e *E* de duas instruções distintas. Esse esquema pode resultar em uma taxa de execução de instruções até duas vezes maior do que no caso de execução seqüencial das instruções. Dois problemas evitam que a execução das instruções possa ser feita em um tempo mínimo. O primeiro é que supomos que uma memória com uma

única porta esteja sendo usada, o que permite apenas um acesso à memória por fase. Isso requer a inserção de um estado de espera em algumas instruções. O segundo é que uma instrução de desvio interrompe o fluxo seqüencial de execução. Para resolver esse problema com um mínimo de circuitos adicionais, o compilador ou montador pode inserir uma instrução NOOP no fluxo de instrução.

O desempenho da *pipeline* pode ser melhorado ainda mais se forem possíveis dois acessos à memória por fase. Isso produz a seqüência mostrada na Figura 12.6c. Nesse caso, até três instruções podem ser sobrepostas e a melhoria da taxa de execução de instruções é de, no máximo, três vezes. Novamente, instruções de desvio provocam um aumento no tempo de execução de instruções para um valor maior que o mínimo possível. Além disso, note que dependências de dados também têm algum efeito. Se uma instrução requer um operando que é alterado pela instrução anterior, é necessário um atraso. Mais uma vez, isso pode ser obtido pela introdução de um instrução NOOP.

A *pipeline* discutida até aqui funciona melhor se as três fases são de duração aproximadamente igual. Como a fase *E* normalmente envolve uma operação da ULA, ela pode ter maior duração. Nesse caso, ela pode ser subdividida em duas subfases:

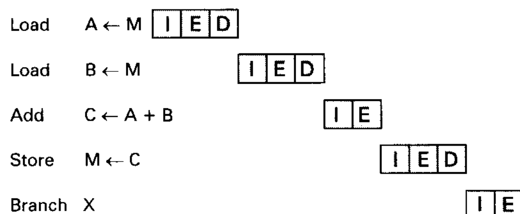
- E_1: Leitura do banco de registradores
- E_2: Operação da ULA e escrita em registrador

Em virtude da simplicidade e da regularidade do conjunto de instruções, a subdivisão do ciclo de execução em três ou quatro fases é feita mais facilmente. A Figura 12.6d mostra o resultado com uma *pipeline* de quatro caminhos. Podem ser executadas até quatro instruções de cada vez, e o ganho de desempenho máximo em termos de tempo de execução é de quatro vezes. Note, mais uma vez, o uso de instruções NOOP para lidar com atrasos devidos a desvios ou dependências de dados.

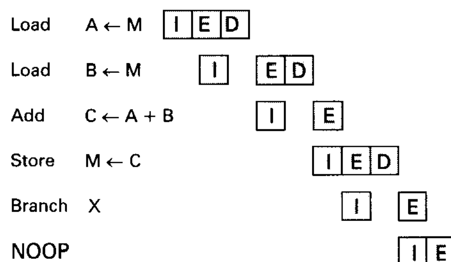
Otimização da *pipeline*

Em razão da natureza simples e regular das instruções RISC, é possível empregar *pipelines* de maneira eficiente. Existem algumas poucas variações na duração da execução de instruções, e a *pipeline* pode ser projetada para refletir isso. Vimos, entretanto, que dependências de dados e desvios reduzem a taxa de execução global.

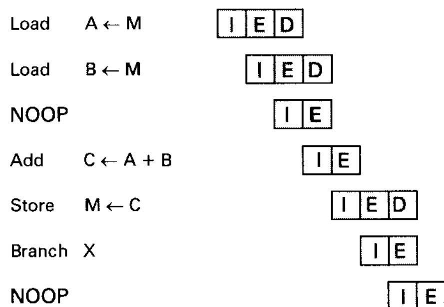
Para compensar essas dependências, são usadas técnicas de reordenação das instruções do código. Primeiramente, consideramos as instruções de desvio. O *desvio atrasado* (*delayed branch*), uma técnica para aumentar a eficiência da *pipeline*, adota um desvio que não tem efeito antes de terminada a execução da instrução seguinte (por isso o termo *atrasado*). A posição da instrução imediatamente seguinte ao desvio é conhecida como *posição de atraso* (*delay slot*). Essa estranha técnica é mostrada na Tabela 12.9. A coluna rotulada como 'desvio normal' mostra um programa escrito em linguagem de máquina com instruções simbólicas usuais. Depois que a instrução 102 é executada, a próxima instrução a ser executada é a 105. Para regularizar o fluxo de instruções na *pipeline*, é inserida uma instrução NOOP após o desvio. Entretanto, um desempenho melhor pode ser obtido se as instruções 101 e 102 são trocadas de posição. A Figura 12.7 mostra o resultado. A instrução JUMP é buscada antes da instrução ADD. Note, entretanto, que a instrução ADD é buscada antes que a execução da instrução JUMP tenha chance de alterar o contador de programa. Portanto, a semântica original do programa é mantida.



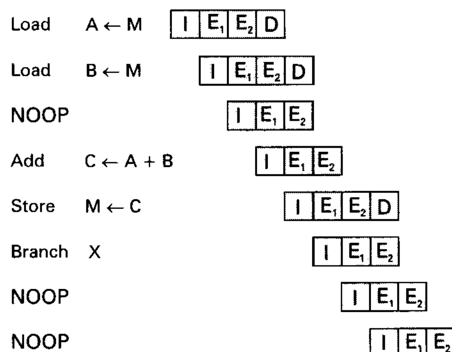
(a) Execução seqüencial



(b) Diagrama *pipeline* com dois caminhos



(c) Diagrama de *pipeline* com três caminhos



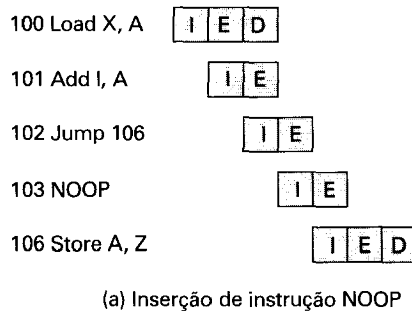
(d) Diagrama de *pipeline* com quatro caminhos

Figura 12.6 Efeito do uso de *pipeline*.

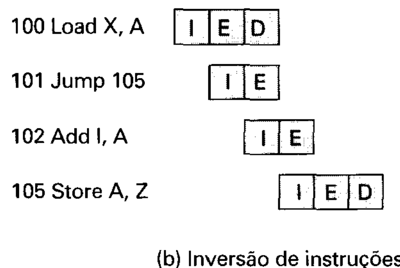
Essa mudança na ordem de instruções tem sucesso no caso de desvios incondicionais e de chamadas e retornos de procedimentos. No caso de desvios condicionais, esse procedimento não pode ser cegamente aplicado. Se a condição testada para fazer o desvio pode ser alterada pela instrução imediatamente precedente, o compilador não deve efetuar a troca de posição das instruções, ficando limitado a inserir uma instrução NOOP. Alternativamente, o compilador pode procurar inserir uma instrução útil após a instrução de desvio. A experiência com os sistemas RISC de Berkeley e IBM 801 mostra que a maioria das instruções de desvio condicional pode ser otimizada desse modo (Patterson, 1982a; Radin, 1983).

Tabela 12.9 Desvio normal e atrasado

Endereço	Desvio normal		Desvio atrasado		Desvio atrasado otimizado	
100	LOAD	X,A	LOAD	X,A	LOAD	X,A
101	ADD	1,A	ADD	1,A	JUMP	105
102	JUMP	105	JUMP	106	ADD	1,A
103	ADD	A,B	NOOP		ADD	A,B
104	SUB	C,B	ADD	A,B	SUB	C,B
105	STORE	A,Z	SUB	C,B	STORE	A,Z
106			STORE	A,Z		



(a) Inserção de instrução NOOP



(b) Inversão de instruções

Figura 12.7 Uso de desvio atrasado.

Uma estratégia semelhante, denominada *carga atrasada* (*delayed load*), pode ser usada para instruções LOAD. Em uma instrução LOAD, o registrador-alvo da carga é bloqueado pelo processador. O processador continua, então, a execução do fluxo de instruções, até encontrar uma instrução que use esse registrador, e, nesse ponto, ele passa a ficar inativo, até que a carga do valor no registrador seja completada. Se o compilador puder reordenar instruções de modo que possa ser realizado algum trabalho útil enquanto a instrução de carga permanece na *pipeline*, a eficiência da execução de instruções será aumentada.

Como observação final, devemos notar que o projeto da *pipeline* de instruções deve também levar em conta outras técnicas de otimização aplicadas ao sistema. Por exemplo, Bradlee (1991b) mostra que o escalonamento de instruções para execução na *pipeline* e a alocação dinâmica de registradores devem ser considerados em conjunto, para se obter maior eficiência.

12.6 MIPS R4000

Um dos primeiros conjuntos de processadores RISC disponíveis comercialmente foi desenvolvido pela MIPS Technology Inc. O sistema foi inspirado em um sistema experimental, também denominado MIPS, desenvolvido na Universidade de Stanford (Hennessy, 1984) nos Estados Unidos. Nesta seção abordamos o processador MIPS R4000. Ele tem essencialmente a mesma arquitetura e conjunto de instruções dos projetos MIPS anteriores: o R2000, o R3000 e o R6000. A diferença mais significativa é que o R4000 usa 64 bits, e não 32 bits, para todos os caminhos de dados internos e externos, assim como para endereços, registradores e para a ULA.

O uso de 64 bits tem diversas vantagens sobre uma arquitetura de 32 bits. Possibilita maior espaço de endereçamento suficiente para que o sistema operacional possa mapear mais

de um terabyte de arquivos diretamente na memória virtual, para fácil acesso. Sendo atualmente comuns discos de 1 ou mais gigabytes, o espaço de endereçamento de 4 gigabytes de uma máquina de 32 bits torna-se um fator restritivo. Além disso, o uso de 64 bits possibilita ao R4000 processar dados como números de ponto flutuante de precisão simples no padrão IEEE e seqüências de caracteres, processando até oito caracteres em uma única ação.

A pastilha do processador R4000 é dividida em duas seções, uma contém a CPU e outra um co-processador para gerenciamento de memória. O processador tem uma arquitetura muito simples. A intenção foi projetar um sistema em que a lógica de execução de instruções fosse a mais simples possível, deixando espaço disponível para lógicas de otimização de desempenho (por exemplo, toda a unidade de gerenciamento de memória).

O processador contém 32 registradores de 64 bits. Ele também contém até 128 Kbytes de memória cache de alta velocidade, metade dos quais é alocada para instruções e a outra para dados. A cache relativamente grande (o IBM 3090 possui de 128 a 256 Kbytes de cache) possibilita ao sistema manter grandes conjuntos de código e dados locais de programas para uso direto pelo processador, diminuindo a sobrecarga do barramento da memória principal e evitando a necessidade de usar um grande banco de registradores, com sua respectiva lógica para controle de janelas de registradores.

Conjunto de instruções

A Tabela 12.10 apresenta o conjunto básico de instruções dos processadores da série MIPS R. A Tabela 12.11 inclui instruções adicionais implementadas no R4000. Todas as instruções do processador são codificadas em um único formato de palavra de 32 bits. Todas as operações de dados são de registrador para registrador; as únicas referências à memória são em operações puramente de carga ou armazenamento.

O R4000 não usa códigos de condição. Se uma instrução gera uma dada condição, o padrão de bits correspondente é armazenado em um registrador de propósito geral. Isso evita o uso de lógica especial para lidar com códigos de condição, uma vez que elas afetam o mecanismo de *pipeline* de instruções e a reordenação de instruções pelo compilador. Em vez disso, é empregado o mecanismo já existente para lidar com dependências de valores em registradores. Além disso, essas condições mapeadas nos bancos de registradores estão sujeitas às mesmas otimizações aplicadas pelo compilador para alocação e reutilização dos demais valores armazenados em registradores.

Assim como acontece com a maioria das máquinas baseadas em RISC, o MIPS usa um único tamanho de instrução de 32 bits. Esse tamanho único de instrução simplifica a busca e a decodificação de instruções, assim como a interação da busca de instruções com a unidade de gerenciamento de memória virtual (por exemplo, instruções que não cruzam limites de palavras ou de páginas). Os três formatos de instrução (Figura 12.8) compartilham uma formatação comum de códigos de operação e referências a registradores, simplificando a decodificação de instruções. O efeito de instruções mais complexas é sintetizado em tempo de compilação.

Tabela 12.10 Conjunto de instrução da série MIPS R

OP	Descrição	OP	Descrição
	Instruções de carga/armazenamento		**Instruções de divisão/multiplicação**
LB	Carregar byte	MULT	Multiplicação
LBU	Carregar byte sem sinal	MULTU	Multiplicação sem sinal
LH	Carregar meia palavra	DIV	Divisão
LHU	Carregar meia palavra sem sinal	DIVU	Divisão sem sinal
LW	Carregar palavra	MFHI	Mover de HI
LWL	Carregar palavra da esquerda	MTHI	Mover para HI
LWR	Carregar palavra da direita	MFLO	Mover de LO
SB	Armazenar byte	MTLO	Mover para LO
SH	Armazenar meia palavra		**Instruções de desvio**
SW	Armazenar palavra	J	Desvio incondicional
SWL	Armazenar palavra à esquerda	JAL	Desvio incondicional com ligação
SWR	Armazenar palavra à direita	JR	Desvio incondicional para registrador
	Instruções aritméticas (ULA)	JALR	Desvio incondicional para registrador com ligação
ADDI	Adicionar operando imediato	BEQ	Desvio se igual
ADDIU	Adicionar operando imediato sem sinal	BNE	Desvio se diferente
SLTI	Atribuir 1 se menor que operando imediato	BLEZ	Desvio se menor ou igual a zero
SLTIU	Atribuir 1 se menor que operando imediato sem sinal	BGTZ	Desvio se maior que zero
ANDI	AND com operando imediato	BLTZ	Desvio se menor que zero
ORI	OR com operando imediato	BGEZ	Desvio se maior ou igual a zero
XORI	XOR com operando imediato	BLTZAL	Desvio se menor que zero com ligação
LUI	Carregar metade superior com operando imediato	BGEZAL	Desvio se maior ou igual a zero com ligação
	Instruções aritméticas (3 operandos, tipo R)		**Instruções do co-processador**
ADD	Adicionar	LWCz	Carregar palavra para co-processador
ADDU	Adicionar sem sinal	SWCz	Armazenar palavra de co-processador
SUB	Subtrair	MTCz	Mover para co-processador
SUBU	Subtrair sem sinal	MFCz	Mover do co-processador
SLT	Atribuir 1 se menor que	CTCz	Mover controle para co-processador
SLTU	Atribuir 1 se menor que	CFCz	Mover controle do co-processador
AND	AND	COPz	Operação do co-processador
OR	OR	BCzT	Desvio se condição z do co-processador é verdadeira
XOR	XOR	BCzF	Desvio se condição z do co-processador é falsa
NOR	NOR		**Instruções especiais**
	Instruções de deslocamento	SYSCALL	Chamada de sistema
SLL	Deslocamento lógico para esquerda	BREAK	Gera uma exceção
SRL	Deslocamento lógico para direita		
SRA	Deslocamento aritmético para direita		
SLLV	Deslocamento lógico variável para esquerda		
SRLV	Deslocamento lógico variável para direita		
SRAV	Deslocamento aritmético variável para direita		

Tabela 12.11 Instruções adicionais do R4000

OP	Descrição	OP	Descrição
	Instruções de carga/armazenamento		**Instruções de exceção**
LL	Carga ligada	TGE	Interrompe se maior ou igual
SC	Armazenamento condicional	TLGEU	Interrompe se maior ou igual, sem sinal
SYNC	Sincronização	TLT	Interrompe se menor
	Instruções de desvio	TLTU	Interrompe se menor, sem sinal
BEQL	Desvio provável se igual	TEQ	Interrompe se igual
BNEL	Desvio provável se diferente	TNE	Interrompe se diferente
BLEZL	Desvio provável se menor ou igual a zero	TGEI	Interrompe se maior ou igual a operando imediato
BGTZL	Desvio provável se maior que zero	TGEIU	Interrompe se maior ou igual a operando imediato, sem sinal
BLTZL	Desvio provável se menor que zero		
BGEZL	Desvio provável se maior ou igual a zero	TLTI	Interrompe se menor que operando imediato
		TLTIU	Interrompe se menor que operando imediato, sem sinal
BLTZALL	Desvio provável se menor que zero com ligação	TEQI	Interrompe se igual a operando imediato
BGEZALL	Desvio provável se maior ou igual a zero com ligação	TNEI	Interrompe se diferente de operando imediato
			Instruções de co-processador
BCzTL	Desvio provável se condição z do co-processador verdadeira	LDCz	Carregar palavra dupla para co-processador
		SDCz	Armazenar palavra dupla de co-processador
CDzFL	Desvio provável se condição z do co-processador falsa		

Apenas o modo mais simples e mais freqüentemente usado de endereçamento à memória é implementado no hardware. Todas as referências à memória consistem em um deslocamento de 16 bits, relativo ao conteúdo de um registrador de 32 bits. Por exemplo, a instrução para 'carregar palavra' tem a forma:

```
lw r2, 128 (r3) carregar no registrador r2 a palavra cujo endereço é um
   deslocamento de 128 bytes, a partir do conteúdo do registrador r3
```

Cada um dos 32 registradores de propósito geral pode ser usado como o registrador base. O registrador r0 contém sempre o valor 0.

Para sintetizar outros modos de endereçamento típicos de máquinas convencionais, o compilador usa múltiplas instruções de máquina. Alguns exemplos são apresentados na Tabela 12.12 (Chow, 1987). A tabela mostra o uso de instruções `lui` (carregar metade superior de operando imediato — *load upper immediate*), que carrega a metade superior de um registrador com um valor imediato de 16 bits, atribuindo valor zero à metade inferior.

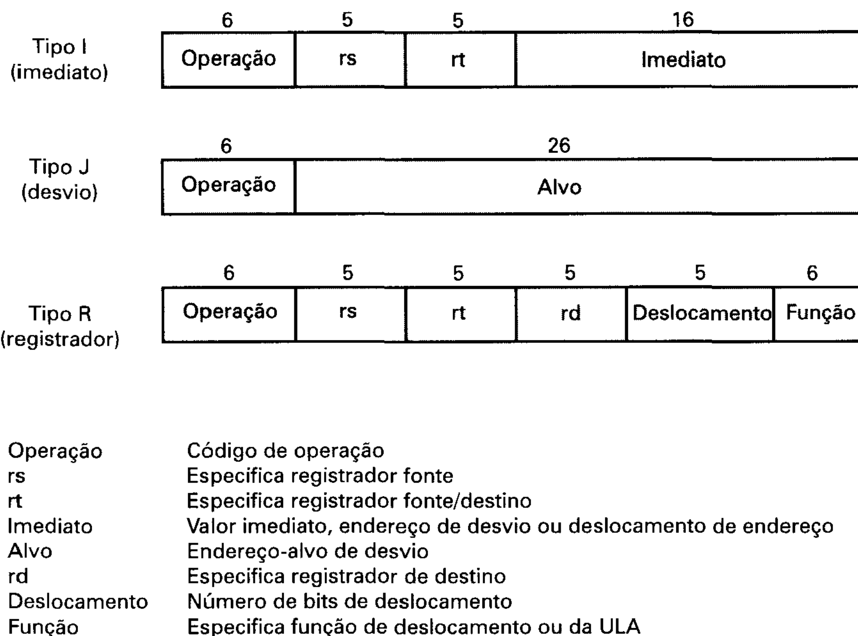


Figura 12.8 Formatos de instrução do MIPS.

Pipeline de instruções

Com sua arquitetura de instrução simplificada, o MIPS pode fazer uso bastante eficiente da *pipeline*. É instrutivo examinar a evolução da *pipeline* na arquitetura MIPS, uma vez que ela mostra a evolução de *pipelines* RISC em geral.

Os primeiros sistemas RISC experimentais e a primeira geração de processadores RISC comerciais tinham uma taxa de execução de aproximadamente uma instrução por ciclo de relógio do sistema. Para melhorar esse desempenho, foram projetadas duas classes de processadores, destinadas a possibilitar a execução de várias instruções por ciclo de relógio: as arquiteturas superescalares e as arquiteturas de superpipeline. Uma arquitetura superescalar essencialmente replica cada um dos estágios da *pipeline*, possibilitando que duas ou mais instruções em um mesmo estágio da *pipeline* possam ser processadas simultaneamente. Uma arquitetura de superpipeline é um refinamento da estrutura da *pipeline*, que usa maior número de estágios. Com um número de estágios maior, mais instruções podem estar na *pipeline* ao mesmo tempo, aumentando o paralelismo.

Tabela 12.12 Sintetizando outros modos de endereçamento usando o modo de endereçamento do MIPS

Instrução aparente	**Instrução real**
lw r2,<deslocamento de 16 bits>	lw r2,<deslocamento de 16 bits> (r0)
lw r2, <deslocamento de 32 bits>	lui r1, <metade superior do deslocamento> lw r2, <metade inferior do deslocamento> (r1)
lw r2, <deslocamento de 32 bits> (r4)	lui r1, <metade superior do deslocamento> addu r1, r1, r4 lw r2, <metade inferior do deslocamento> (r1)

As duas abordagens possuem limitações. Em uma *pipeline* superescalar, dependências entre instruções que são executadas em *pipelines* diferentes podem retardar o sistema. Além disso, uma lógica especial é necessária para organizar o fluxo de instruções na *pipeline*, de modo que coordene essas dependências. Em uma superpipeline, existe uma sobrecarga associada à transferência de instruções de um estágio para o seguinte.

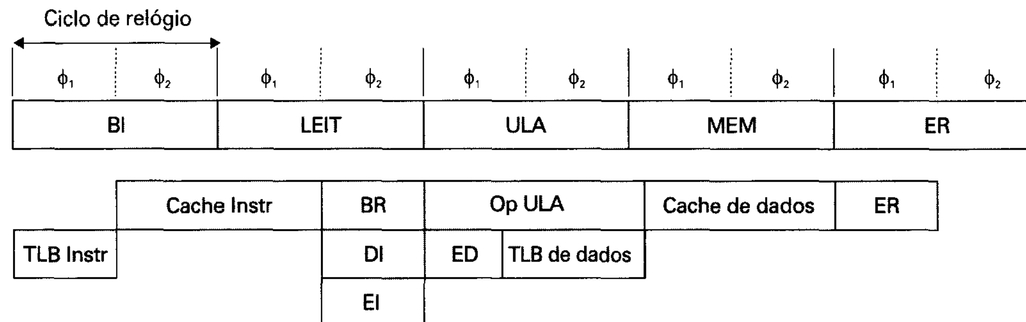
O Capítulo 13 é dedicado às arquiteturas superescalares. O MIPS R4000 é um bom exemplo de arquitetura de superpipeline baseada em RISC.

A Figura 12.9a mostra a *pipeline* de instruções do R3000. No R3000, a *pipeline* avança uma vez por ciclo de relógio. O compilador MIPS é capaz de reordenar instruções de modo que preencha posições de atraso, em 70% a 90% do tempo. Todas as instruções seguem a mesma seqüência de cinco estágios na *pipeline*:

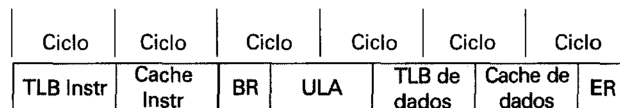
- Busca da instrução
- Busca do operando fonte no banco de registradores
- Operação da ULA ou geração de endereço de operando
- Referência ao dado na memória
- Armazenamento do resultado no banco de registradores

Como mostra a Figura 12.9a, existe paralelismo não apenas em virtude da *pipeline* mas também na execução de uma única instrução. O ciclo de relógio de 60 ns é dividido em duas fases de 30 ns. Operações de acesso a instruções e dados na cache externa requerem, cada uma, 60 ns, assim como as principais operações internas (OP, DA, IA). A decodificação de instrução é uma operação mais simples, requerendo uma única fase de 30 ns, e é sobreposta com a busca por um registrador usado na mesma instrução. O cálculo do endereço de uma instrução de desvio também se sobrepõe à decodificação de instrução e busca de registrador, de modo que um desvio na instrução *i* possa endereçar o acesso à cache de instruções (ICACHE) da instrução *i* + 2. Analogamente, uma operação de carga na instrução *i* busca dados que são imediatamente usados pela operação da instrução *i* + 1 e o resultado de uma operação de deslocamento ou da ULA é passado diretamente para a instrução *i* + 1, sem nenhum atraso. Esse forte acoplamento entre instruções resulta em uma *pipeline* altamente eficiente.

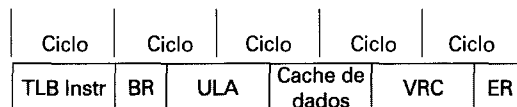
Mais detalhadamente, cada ciclo de relógio é dividido em fases separadas, simbolizadas por ø1 e ø2. As funções executadas em cada fase são resumidas na Tabela 12.13.



(a) *Pipeline* do R3000 detalhada



(b) *Pipeline* R3000 modificada, com latências reduzidas



(c) *Pipeline* R3000 otimizada com acessos paralelos à TLB e à cache

BI	= busca de instrução
LEIT	= leitura
MEM	= acesso à memória
ER	= escrita de resultado
Cache Instr	= acesso à cache de instrução
BR	= busca de operando em registrador
Cache de dados	= acesso à cache de dados
TLB Instr	= tradução de endereço de instrução
DI	= decodificação de instrução
EI	= cálculo de endereço de instrução
ED	= cálculo de endereço virtual de dado
TLB de dados	= tradução de rótulos de cache de dados
VRC	= verificação de rótulos de cache de dados

Figura 12.9 Melhorando a *pipeline* do R3000.

O R4000 incorpora diversos avanços tecnológicos em relação ao R3000. O uso de tecnologia mais avançada permitiu a diminuição do ciclo de relógio para a metade (30 ns). Além disso, a densidade maior da pastilha possibilitou que as caches de instrução e de dados fossem incorporadas à pastilha do processador. Antes de examinar a *pipeline* do R4000, consideramos como a *pipeline* do R3000 pode ser modificada para melhorar o desempenho, usando a tecnologia do R4000.

Um primeiro passo é mostrado na Figura 12.9b. Lembre-se de que cada ciclo nessa figura tem a metade do tempo do ciclo da Figura 12.9a. Como as caches de instrução e de dados estão na mesma pastilha do processador, os estágios de acesso a essas caches requerem apenas meio ciclo: assim eles ainda consomem apenas um ciclo de relógio. Em virtude da velocidade de acesso ao banco de registradores, a leitura e a escrita em registrador continuam consumindo apenas metade de um ciclo de relógio.

Tabela 12.13 Estágios da *pipeline* do R3000

Estágio da *pipeline*	**Fase**	**Função**
BI	$\phi 1$	Traduz um endereço virtual de instrução para um endereço físico (depois de uma decisão de desvio) usando a TLB.
BI	$\phi 2$	Envia endereço físico para a cache de instrução.
LEIT	$\phi 1$	Retorna instrução da cache de instrução. Compara rótulo e validade da instrução buscada.
LEIT	$\phi 2$	Decodifica instrução. Busca operando no banco de registradores. Em caso de desvio, calcula o endereço-alvo do desvio.
ULA	$\phi 1 + \phi 2$	Em caso de operação registrador para registrador, executa a operação aritmética ou lógica.
ULA	$\phi 1$	Em caso de desvio condicional, decide se o desvio será tomado ou não. Em caso de referência à memória (carga ou armazenamento), calcula o endereço virtual do dado.
ULA	$\phi 2$	Em caso de referência à memória, traduz o endereço virtual do dado para o endereço físico usando a TLB.
MEM	$\phi 1$	Se for uma referência à memória, envia endereço físico para a cache de dados.
MEM	$\phi 2$	Em caso de referência à memória, retorna o dado da cache de dados e verifica rótulos.
ER	$\phi 1$	Escreve dado no banco de registradores.

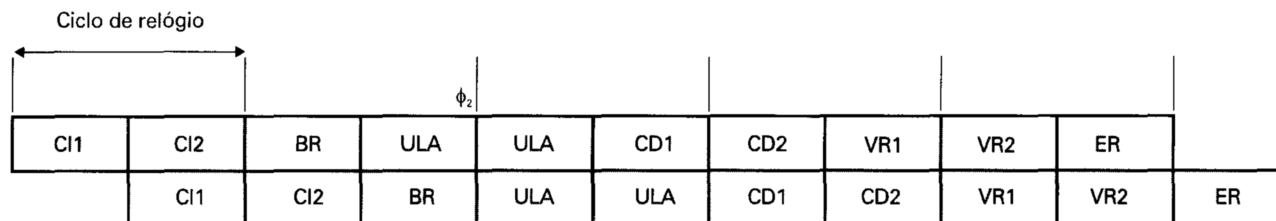
Como as caches do R4000 são internas à pastilha, a tradução de endereço virtual para endereço físico pode atrasar o acesso à cache. Esse atraso é reduzido com a implementação de caches indexadas com endereços virtuais e com a execução da tradução de endereço e do acesso à cache em paralelo. A Figura 12.9c mostra a *pipeline* do R3000 com essa otimização. Em virtude da compressão desses eventos, a verificação de rótulos da cache de dados é feita separadamente, no ciclo seguinte ao de acesso à cache.

Em um sistema com superpipeline, o hardware existente é usado várias vezes por ciclo, inserindo-se registradores de *pipeline* para subdividir cada estágio. Essencialmente, cada estágio da *pipeline* opera em uma freqüência múltipla da freqüência básica do relógio, onde esse múltiplo depende do grau de superpipeline. A tecnologia do R4000 tem velocidade e densidade de pastilha que possibilitam uma superpipeline de grau 2. A Figura 12.10a mostra a *pipeline* R3000 otimizada, usando esse grau de superpipeline. Note que essa *pipeline* tem, essencialmente, a mesma estrutura dinâmica apresentada na Figura 12.9c.

Outras melhorias podem ser introduzidas. Para o R4000, foi projetado um somador especializado, muito mais largo. Isso torna possível executar operações da ULA com velocidade duas vezes maior. Outra melhoria possibilita a execução de carga e armazenamento com velocidade com o dobro da velocidade. A *pipeline* resultante é mostrada na Figura 12.10b.

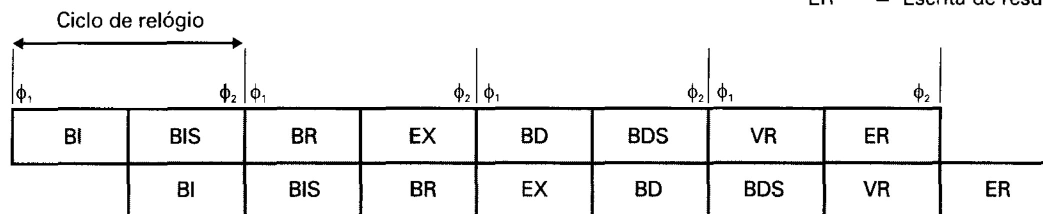
A *pipeline* do R4000 tem oito estágios, o que significa que até oito instruções podem estar executando na *pipeline* ao mesmo tempo. A *pipeline* avança a uma taxa de dois estágios por ciclo de relógio. Os oito estágios da *pipeline* são:

- **Primeira metade de busca de instrução:** o endereço virtual é apresentado à cache de instruções e à cache de tradução de endereços (TLB).
- **Segunda metade de busca de instrução:** a cache de instruções retorna a instrução e a TLB gera o endereço físico.
- **Conjunto de registradores:** três atividades ocorrem em paralelo:
  - A instrução é decodificada e são verificadas condições de bloqueio mútuo entre instruções (por exemplo, se a instrução depende do resultado de uma instrução precedente).
  - É efetuada a verificação de rótulo na cache de instrução.
  - Os operandos são buscados no banco de registradores.
- **Execução de instrução:** uma das três atividades pode ocorrer:
  - Se a instrução é uma operação registrador para registrador, a ULA executa a operação aritmética ou lógica.
  - Se a instrução é de carga ou armazenamento, o endereço virtual do dado é calculado.
  - Se a instrução é um desvio, o endereço virtual do alvo do desvio é calculado e as condições de desvio são verificadas.
- **Primeiro estágio da cache de dados:** o endereço virtual é apresentado à cache de dados e à TLB.
- **Segundo estágio da cache de dados:** a cache de dados retorna o dado e a TLB gera o endereço físico.
- **Verificação de rótulos:** é efetuada a verificação de rótulos da cache para instruções de carga e armazenamento.
- **Escrita de resultado:** o resultado da instrução é escrito no banco de registradores.



(a) Implementação otimizada da *pipeline* do R3000 como uma *superpipeline*

BI = Primeira metade da busca de instrução
BIS = Segunda metade da busca de instrução
BR = Busca de operandos em registradores
EX = Execução de instrução
CI = Cache de instrução
CD = Cache de dados
BD = Primeira metade de cache de dados
BDS = Segunda metade de cache de dados
VR = Verificação de rótulos
ER = Escrita de resultados



(b) *Pipeline* do R4000

Figura 12.10 Superpipeline teórica no R3000 e *superpipeline* real no R4000.

12.7 SPARC

O termo SPARC (*scalable processor architecture* — arquitetura de processadores escaláveis) refere-se a uma arquitetura definida pela Sun Microsystems. A Sun desenvolveu sua própria implementação SPARC, mas também licenciou a arquitetura para outros fabricantes, para que produzissem máquinas compatíveis com essa arquitetura. A arquitetura SPARC é inspirada na máquina RISC I de Berkeley, e seu conjunto de instruções e organização de registradores é fortemente baseado no modelo RISC de Berkeley.

Conjunto de registradores da SPARC

Assim como a arquitetura RISC de Berkeley, a arquitetura SPARC usa janelas de registradores. Cada janela consiste em 24 registradores, e o número total de janelas é dependente de implementação, variando entre 2 e 32 janelas. A Figura 12.11 apresenta uma implementação com oito janelas, usando um total de 136 registradores físicos; como indica a discussão apresentada na Seção 12.1, esse número de janelas parece ser razoável. Os registradores físicos 0 a 7 são registradores globais, compartilhados por todos os procedimentos. Cada processo vê registradores lógicos numerados de 0 a 31. Os registradores lógicos 24 a 31, conhecidos como *ins*, são compartilhados com o procedimento *pai*, que efetuou a chamada; os registradores lógicos 8 a 15, conhecidos como *outs*, são compartilhados com qualquer procedimento *filho* ou procedimento chamado. Essas duas partes sobrepõem-se às outras janelas. Os registradores lógicos 16 a 23, conhecidos como *locais*, não são compartilhados e não se sobrepõem a outras janelas. Como também indica a discussão apresentada na Seção 4.1, o uso de oito registradores para passagem de parâmetros parece ser adequado na maioria dos casos (por exemplo, veja a Tabela 12.4).

A Figura 12.12 mostra outra visão da sobreposição de registradores. O procedimento que efetua a chamada coloca os parâmetros a serem passados em seus registradores *outs*; o procedimento chamado trata esses mesmos registradores físicos como os seus registradores *ins*. O processador mantém um apontador de janela corrente (CWP), localizado no registrador de estado do processador (*processor status register* — PSR), que aponta para a janela do procedimento que está sendo executado. Uma máscara de invalidação de janela (*window invalid mask* — WIM), também localizada no PSR, indica quais janelas são inválidas.

Com a arquitetura de registradores SPARC, normalmente não é necessário armazenar e restaurar registradores em uma chamada de procedimento. O compilador é simplificado porque apenas tem de se preocupar em alocar registradores locais para o procedimento de modo eficiente, não precisando preocupar-se com a alocação de registradores entre procedimentos.

Conjunto de instruções

A Tabela 12.14 enumera as instruções da arquitetura SPARC. A maioria das instruções usa apenas operandos em registradores. Instruções de registrador para registrador têm três operandos e podem ser expressas na forma:

$$R_d \leftarrow R_{S1} \text{ op } S2$$

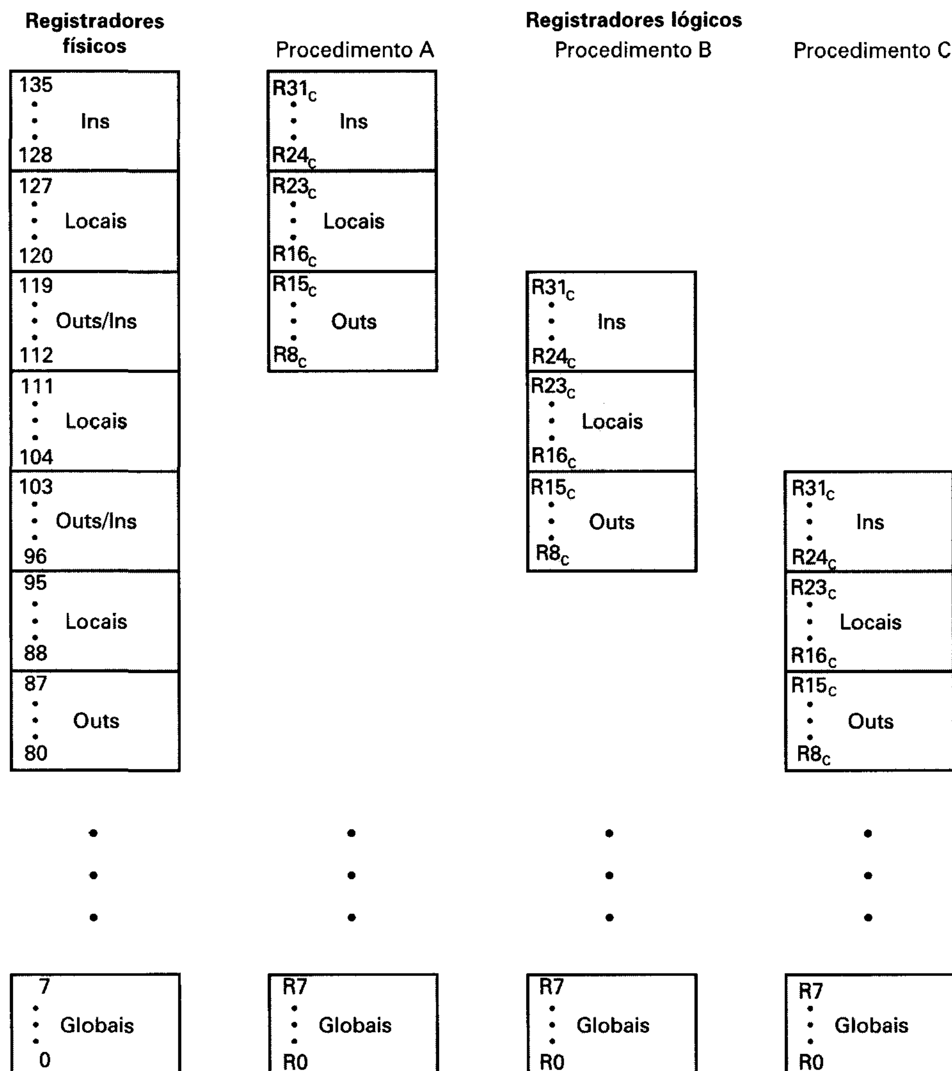


Figura 12.11 Esquema de janelas de registradores do SPARC para três procedimentos.

R_d e R_{s1} são referências a registradores; S2 pode referenciar um registrador ou um operando imediato de 13 bits. O registrador zero (R_0) é mantido sempre com o valor 0 pelo hardware. Essa forma é bastante adequada para programas típicos, que têm alta proporção de constantes e variáveis locais escalares.

As operações disponíveis na ULA podem ser agrupadas do seguinte modo:

- Adição inteira (com ou sem 'vai-um')
- Subtração inteira (com ou sem 'vai-um')
- Operações booleanas bit a bit AND, OR, XOR e suas negações
- Deslocamento lógico para a esquerda, deslocamento lógico para a direita e deslocamento aritmético para a direita

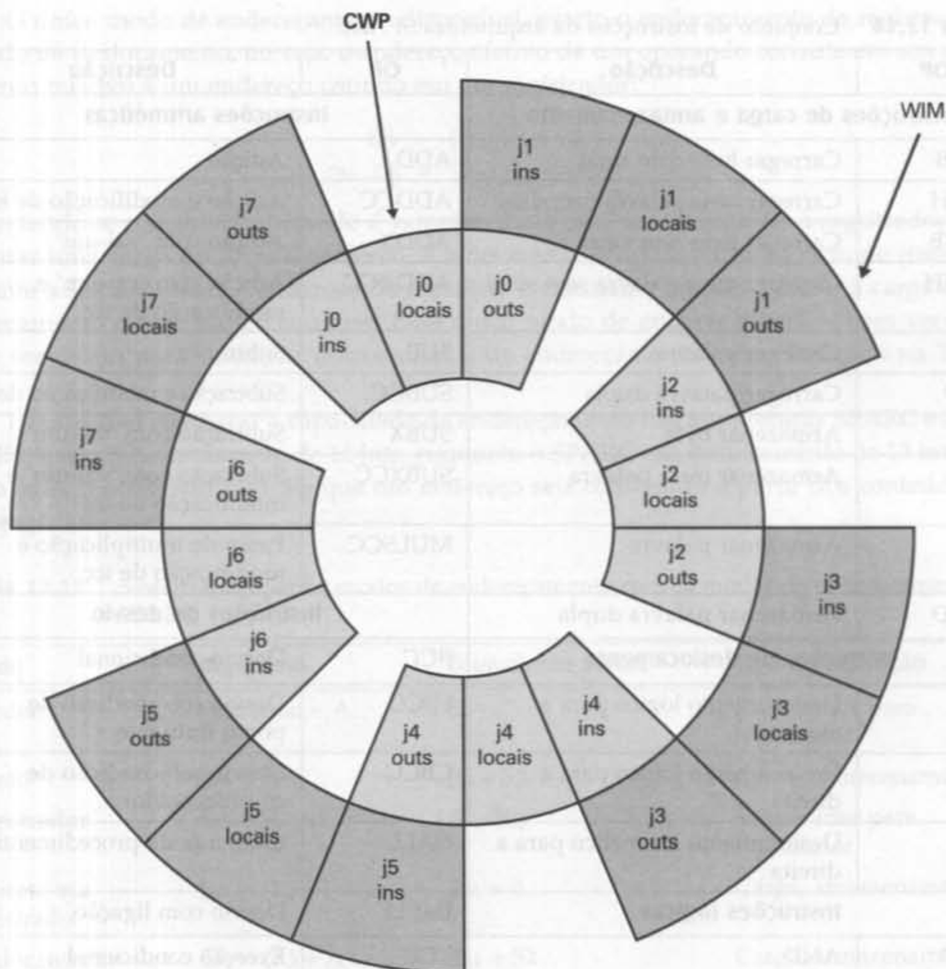


Figura 12.12 Oito janelas de registradores formando uma pilha circular na arquitetura SPARC.

Todas essas instruções, exceto os deslocamentos, podem opcionalmente modificar os quatro códigos de condição (ZERO, NEGATIVO, OVERFLOW, 'VAI-UM'). Números inteiros são representados em complemento de dois com 32 bits.

Apenas instruções simples de carga e armazenamento referenciam a memória. Existem instruções distintas para carregar e armazenar palavra (32 bits), palavra dupla, meia palavra e byte. Para os dois últimos casos, existem instruções para carregar números com ou sem sinal. Números com sinal são estendidos para preencher o registrador de destino de 32 bits. Números sem sinal são completados com zero.

Tabela 12.14 Conjunto de instruções da arquitetura SPARC

OP	Descrição	OP	Descrição
Instruções de carga e armazenamento		**Instruções aritméticas**	
LSDB	Carregar byte com sinal	ADD	Adição
LDSH	Carregar meia palavra com sinal	ADDCC	Adição e modificação de icc
LDUB	Carregar byte sem sinal	ADDX	Adição com 'vai-um'
LDUH	Carregar meia palavra sem sinal	ADDXCC	Adição com 'vai-um' e modificação de icc
LD	Carregar palavra	SUB	Subtração
LDD	Carregar palavra dupla	SUBCC	Subtração e modificação de icc
STB	Armazenar byte	SUBX	Subtração com 'vai-um'
STH	Armazenar meia palavra	SUBXCC	Subtração com 'vai-um' e modificação de icc
STD	Armazenar palavra	MULSCC	Passo de multiplicação e modificação de icc
STDD	Armazenar palavra dupla	**Instruções de desvio**	
Instruções de deslocamento		BCC	Desvio condicional
SLL	Deslocamento lógico para a esquerda	FBCC	Desvio sob condição de ponto flutuante
SRL	Deslocamento lógico para a direita	CBCC	Desvio sob condição de co-processador
SRA	Deslocamento aritmético para a direita	CALL	Chamada de procedimento
Instruções lógicas		JMPL	Desvio com ligação
AND	AND	TCC	Exceção condicional
ANDCC	AND e modificação de icc	SAVE	Avança janela de registradores
ANDN	NAND	RESTORE	Volta janela de registradores
ANDNCC	NAND e modificação de icc	RETT	Retorna de exceção
OR	OR	**Outras instruções**	
ORCC	OR e modificação de icc	SETHI	Modifica os 22 bits mais significativos
ORN	NOR	UNIMP	Instrução não implementada
ORNCC	NOR e modificação de icc	RD	Leitura de registrador especial
XOR	XOR	WR	Escrita em registrador especial
XORCC	XOR e modificação de icc	IFLUSH	Limpa cache de instrução
XNOR	XNOR		
XNORCC	XNOR e modificação de icc		

O único modo de endereçamento disponível, exceto o endereçamento de registrador, é o modo de deslocamento, ou seja, o endereço efetivo de um operando consiste em um deslocamento relativo a um endereço contido em um registrador:

$$EA = (R_{S1}) + S2$$
$$\text{ou } EA = (R_{S1}) + (R_{S2})$$

dependendo se o segundo operando é valor imediato ou é uma referência a registrador. Para executar uma carga ou armazenamento, é adicionada uma fase extra ao ciclo de instrução. Durante a segunda fase, o endereço de memória é calculado, usando a ULA; a carga ou armazenamento ocorre na terceira fase. Esse único modo de endereçamento é bem versátil e pode ser usado para sintetizar outros modos de endereçamento, como indicado na Tabela 12.15.

É instrutivo comparar a capacidade de endereçamento nas arquiteturas SPARC e MIPS. O MIPS usa um deslocamento de 16 bits, enquanto o SPARC usa deslocamento de 13 bits. Por outro lado, o MIPS não permite que um endereço seja construído a partir dos conteúdos de dois registradores.

Tabela 12.15 Sintetizando outros modos de endereçamento com os modos de endereçamento do SPARC

Modo	Algoritmo	Equivalente SPARC	Tipo de instrução
Imediato	operando = A	S2	Registrador para registrador
Direto	EA = A	R_0 + S2	Carga, armazenamento
Registrador	EA = R	R_{S1}, R_{S2}	Registrador para registrador
Indireto via registrador	EA = (R)	R_{S1} + 0	Carga, armazenamento
Deslocamento	EA = (R) + A	R_{S1} + S2	Carga, armazenamento

Formato das instruções

Assim como o MIPS R4000, o SPARC usa um conjunto simples de formatos de instrução de 32 bits (Figura 12.13). Todas as instruções começam com um código de operação de 2 bits. Para certas instruções, esse código é estendido com bits adicionais, em algum outro lugar no formato. Na instrução Call, um operando imediato de 30 bits é estendido com dois bits de valor zero à direita, para formar um endereço relativo ao PC de 32 bits, na representação em complemento de dois. As instruções são alinhadas em limites de 32 bits, de modo que basta essa forma de endereçamento.

A instrução de desvio inclui um campo de condição de 4 bits, correspondente ao código de condição padrão de quatro bits, de maneira que qualquer combinação de condições pode ser testada. O endereço relativo ao PC, de 22 bits, é estendido com dois bits de valor zero à direita, para formar um endereço relativo de 24 bits, em complemento de dois. Um bit especial é usado na instrução Branch, o bit de anulação. Quando esse bit tem valor zero, a instrução seguinte ao desvio sempre é executada, independentemente de o desvio ser tomado ou não.

Essa é uma típica operação de desvio atrasado, encontrada em muitas máquinas RISC e descrita na Seção 12.5 (veja a Figura 12.7). Quando esse bit especial tem valor 1, a instrução seguinte ao desvio somente é executada se o desvio for tomado. O processador suprime o efeito da instrução seguinte ao desvio caso ela já esteja na *pipeline*. Esse bit é útil porque torna mais fácil para o compilador preencher posições de atraso no código gerado, após um desvio condicional. A instrução-alvo do desvio sempre pode ser colocada na posição de atraso, pois, caso o desvio não seja tomado, essa instrução é anulada. A razão da utilidade dessa técnica é que desvios condicionais geralmente são tomados em mais da metade das vezes.

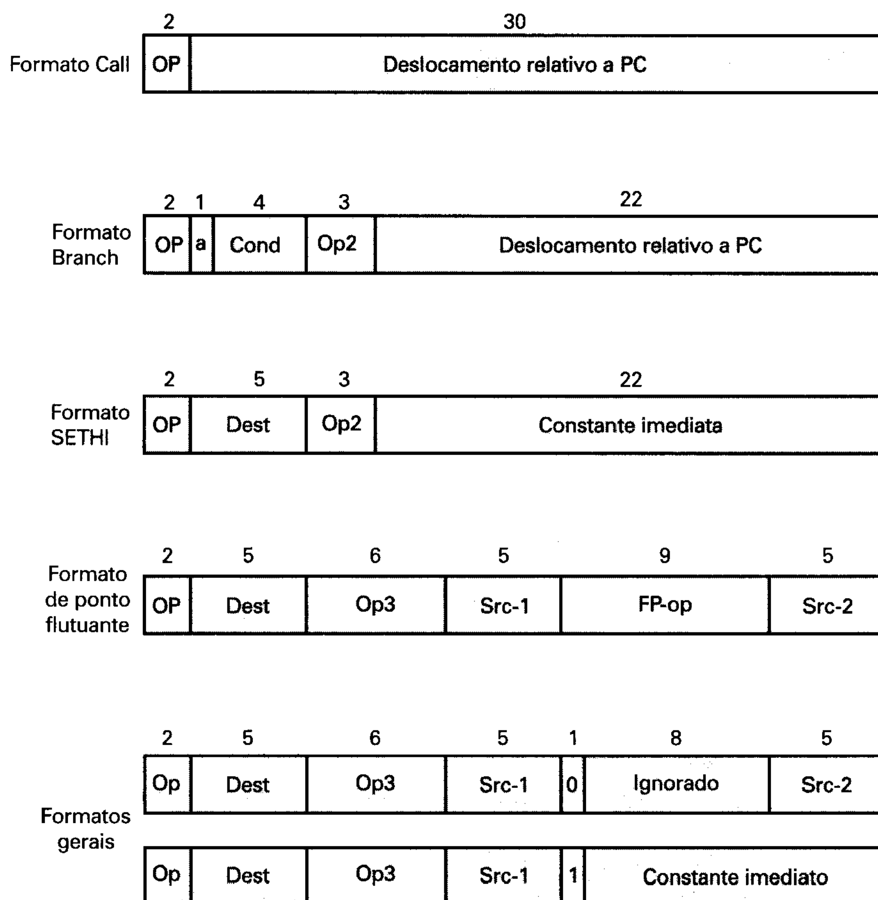


Figura 12.13 Formatos das instruções do SPARC.

A instrução SETHI é uma instrução especial usada para carga ou armazenamento de um valor de 32 bits. Essa característica é necessária para carregar e armazenar endereços e constantes grandes. A instrução SETHI atribui o valor do seu operando imediato de 22 bits aos 22 bits de ordem superior de um registrador, preenchendo com zeros os 10 bits de ordem mais baixa. Uma constante imediata de até 13 bits pode ser especificada em um dos formatos gerais, e esse valor pode ser usado para preencher os 10 bits restantes do registrador. Uma instrução de carga ou armazenamento pode também ser usada para obter o modo de endereça-

mento direto. Para carregar o valor contido na posição K da memória, podem ser usadas as seguintes instruções SPARC:

```
sethi %hi (K), %r8          ;carrega 22 bits de mais alta ordem do
                            ;endereço de memória K no registrador r8
ld  [%r8+%lo(K)], %r8       ;carrega conteúdo da posição K em r8
```

As macros `%hi` e `%lo` são usadas para definir operandos imediatos, que consistem nos bits apropriados do endereço de uma posição. Esse uso de `SETHI` é semelhante ao uso da instrução `LUI` do MIPS (Tabela 12.12).

O formato de ponto flutuante é usado para operações de ponto flutuante. Dois registradores fonte e um registrador destino são especificados.

Finalmente, todas as demais operações, incluindo operações de carga e armazenamento e operações aritméticas e lógicas, usam um dos dois últimos formatos mostrados na Figura 12.13. Um dos formatos usa dois registradores fonte e um registrador destino; o outro formato usa um registrador fonte, um operando imediato de 13 bits e um registrador destino.

12.8 CONTROVÉRSIA RISC *VERSUS* CISC

Por muitos anos, a tendência geral na arquitetura e organização de computadores foi no sentido de aumentar a complexidade do processador: mais instruções, mais modos de endereçamento, maior número de registradores especializados, e assim por diante. O movimento RISC representa uma quebra fundamental na filosofia por trás dessa tendência. Naturalmente, o surgimento de sistemas RISC e a publicação de artigos por seus proponentes exaltando as virtudes da arquitetura RISC levaram a uma reação dos adeptos da que se pode chamar corrente principal da arquitetura de computadores.

Os trabalhos feitos para avaliação dos méritos da abordagem RISC podem ser agrupados em duas categorias:

- **Quantitativos:** buscam comparar o tamanho e a velocidade de execução de programas em máquinas RISC e CISC que usem tecnologia comparável.
- **Qualitativos:** examinam questões tais como suporte a linguagem de alto nível e uso ideal da tecnologia VLSI.

A maioria dos trabalhos de avaliação quantitativa foi feita por pesquisadores que trabalham em sistemas RISC (Patterson, 1982b; Heath, 1984; Patterson, 1984) e tem sido, de longe, favorável à abordagem RISC. Outros examinaram a questão e não saíram convencidos (Colwell, 1985a; Flynn, 1987; Davidson, 1987). Existem diversos problemas ao tentar fazer essas comparações (Serlin, 1986):

- Não existe nenhum par de máquinas RISC e CISC comparáveis em termos de ciclo de vida, custo, nível de tecnologia, complexidade de portas lógicas, sofisticação de compilador, suporte de sistema operacional, e assim por diante.
- Não existe um conjunto de programas de teste definitivo. O desempenho varia com o programa.

- É difícil separar efeitos do hardware e efeitos devidos ao desenvolvimento de compiladores.
- A maioria das análises comparativas de máquinas RISC foi feita em máquinas 'experimentais', e não em produtos comerciais. Além disso, grande parte das máquinas anunciadas como RISC comercialmente disponíveis possui uma mistura de características RISC e CISC. Portanto, fica difícil fazer uma comparação justa com uma máquina comercial 'puramente' CISC (tal como VAX ou Pentium).

A avaliação qualitativa é, quase por definição, subjetiva. Diversos pesquisadores voltaram-se para esse tipo de avaliação (Colwell, 1985a; Wallich, 1985), mas os resultados são, na melhor das hipóteses, ambíguos, e certamente sujeitos a réplica ou contestação (Patterson, 1985b) e, é claro, a contra-réplica (Colwell, 1985b).

Mais recentemente, a controvérsia RISC *versus* CISC diminuiu em grande proporção. Isso porque houve uma convergência gradual das duas tecnologias. Com o aumento da densidade das pastilhas e das velocidades do hardware, os sistemas RISC tornaram-se mais complexos. Ao mesmo tempo, em um esforço para obter o máximo desempenho, projetos CISC focalizaram questões tradicionalmente associadas com as máquinas RISC, tais como um número crescente de registradores de propósito geral e maior ênfase no projeto de *pipeline* de instrução.

12.9 LEITURA RECOMENDADA

Alguns livros-texto que contêm uma boa apresentação sobre conceitos RISC são os de Ward (1990), Patterson (1998) e Hennessy (1996).

As máquinas comerciais MIPS são descritas detalhadamente em Kane (1992). Mirapuri (1992) apresenta uma boa visão do MIPS R4000. A evolução da *pipeline* do R3000 para a superpipeline do R4000 é discutida em Bashteen (1991). A arquitetura SPARC é abordada com algum detalhe em Dewar (1990).

12.10 EXERCÍCIOS

12.1 Considerando o padrão chamada e retorno de procedimentos mostrado na Figura 4.29, quantas situações de *overflow* e *underflow* (cada um dos quais causando uma operação de salvamento/restauração de um registrador) ocorreriam com uma janela de registradores com tamanho de:

a. 5 registradores?
b. 8 registradores?
c. 16 registradores?

12.2 Na discussão sobre a Figura 12.2, foi dito que apenas as primeiras duas partes de uma janela são salvas ou recuperadas. Por que não é necessário armazenar os registradores temporários?

12.3 Desejamos determinar o tempo de execução de um determinado programa usando os vários esquemas de *pipeline* discutidos na Seção 12.5. Suponha que:

N = número de instruções executadas
D = número de acessos à memória
J = número de instruções de desvio

No esquema seqüencial simples (Figura 12.6a), o tempo de execução é de $2N + D$ fases. Obtenha fórmulas para o tempo de execução com *pipelines* de dois, três e quatro caminhos.

12.4 Considere o seguinte fragmento de código de programa em linguagem de alto nível:

```
for I in 1 ... 100 loop
    S ← S + Q(I).VAL
end loop;
```

Suponha que Q é um vetor de registros de 32 bytes e que o campo VAL está nos primeiros 4 bytes de cada registro. Usando o código 80 × 86, podemos compilar esse fragmento de programa do seguinte modo:

```
    MOV   ECX, 1          ;usa registrador ECX para conter I
LP: IMUL  EAX, ECX, 32    ;coloca deslocamento em EAX
    MOV   EBX, Q[EAX]     ;carrega o campo VAL
    ADD   S,  EBX         ;adiciona a S
    INC   ECX             ;incrementa I
    JNE   LP              ;repete até I = 100
```

Esse programa usa a instrução IMUL, que multiplica o segundo operando pelo valor imediato no terceiro operando e coloca o resultado no primeiro operando (ver o Exercício 10.13). Um adepto RISC gostaria de demonstrar que um compilador inteligente pode eliminar instruções complexas desnecessárias, tais como IMUL. Dê uma demonstração disso, reescrevendo o programa 80 × 86 acima sem usar a instrução IMUL.

12.5 Considere o seguinte laço de repetição:

```
S  := 0;
for K  := 1 to  100 do
      S  :=  S - 1;
```

Uma translação direta desse fragmento de programa para uma linguagem de montagem genérica poderia ser:

```
    LD     R1, 0          ;mantém o valor de S em R1
    LD     R2, 1          ;mantém o valor de K em R2
LP: SUB    R1, R1, R2     ;S  := S - 1
    BEQ    R2, 100, FIM   ;termina se K = 100
    ADD    R2, R2, 1      ;senão incrementa K
    JMP    LP             ;voltar ao início do laço de repetição
FIM:
```

Um compilador para uma máquina RISC introduzirá posições de atraso nesse código, de modo que o processador possa empregar o mecanismo de desvio atrasado. É fácil lidar com a instrução JMP, porque ela é sempre seguida pela instrução SUB; portanto, podemos simplesmente colocar uma cópia da instrução SUB na posição de atraso depois de JMP. A instrução BEQ apresenta uma dificuldade. Não podemos deixar o código como está, porque a instrução ADD seria então executada uma vez a mais. Portanto, é necessária uma instrução NOP. Mostre o código resultante.

12.6 Adicione entradas para os seguintes processadores na Tabela 12.8:
- **a.** Pentium II
- **b.** PowerPC

12.7 Em muitos casos, instruções de máquina comuns, que não são relacionadas como parte do conjunto de instruções MIPS, podem ser sintetizadas com uma única instrução MIPS. Mostre que isso ocorre para as seguintes instruções:
- **a.** Mover dado de registrador para registrador
- **b.** Incrementar, decrementar
- **c.** Complementar
- **d.** Negar
- **e.** Limpar

12.8 Uma implementação SPARC tem K janelas de registradores. Qual é o número N de registradores físicos?

12.9 Faltam no conjunto de instruções SPARC diversas instruções comumente encontradas em máquinas CISC. Algumas delas são facilmente simuladas usando o registrador R0, que tem sempre valor igual a 0 ou um operando constante. Essas instruções simuladas são chamadas pseudo-instruções e são reconhecidas pelo compilador SPARC. Mostre como simular as seguintes pseudo-instruções, cada qual com uma única instrução SPARC. Em todos os casos, src e dst referem-se a registradores. (*Dica*: carregar um valor em R0 não efeito.)
- **a.** MOV src, dst
- **b.** COMPARE src1, src2
- **c.** TEST src1
- **d.** NOT dst
- **e.** NEG dst
- **f.** INC dst
- **g.** DEC dst
- **h.** CLEAR dst
- **i.** NOP

12.10 Considere o seguinte fragmento de código:

```
if  K  >  10
    L  := K  + 1
else
    L  := K  - 1;
```

Uma tradução direta desse trecho para código em linguagem de montagem SPARC pode ter a seguinte forma:

```
sethi %hi(K), %r8              ;carrega 22 bits de ordem mais
                               ;data do endereço de posição
                               ;K no registrador r8
ld    [%r8 + %lo(K)], %r8      ;carrega conteúdo da posição K
                               ;em r8
cmp   %r8, 10                  ;compara conteúdo de r8 com 10
ble   L1                       ;desvia se (r8) ≤10
nop
sethi %hi(K), %r9
ld    [%r9 + %lo(K)], %r9      ;carrega o conteúdo da posição
                               ;K em r9
inc   %r                       ;adicionar 1 a (r9)
sethi %hi(L), %r10
st    %r9, [%r10 + %lo(L)]     ;armazena (r9) na posição L
b     L2
nop
```

```
L1: sethi  %hi(K), %r11
    ld     [%r11 + %lo(K)], %r12  ;carrega o conteúdo da posição
                                  ;K em r12
    dec    %r12                   ;subtrai 1 de (r12)
    sethi  %hi(L), %r13
    st     %r12, [%r13 + %lo(L)]  ;armazena (r12) na posição L
L2:
```

O código contém uma instrução nop depois de cada instrução de desvio, para permitir a operação de atraso de desvio.

a. Otimizações de um compilador padrão, que não gera código para máquinas RISC, em geral são efetivas para efetuar duas transformações no código precedente. Note que duas das operações de carga são desnecessárias e duas operações de armazenamento podem ser combinadas se o armazenamento for movido para uma posição diferente no código. Mostre o programa obtido depois de serem feitas essas duas mudanças.

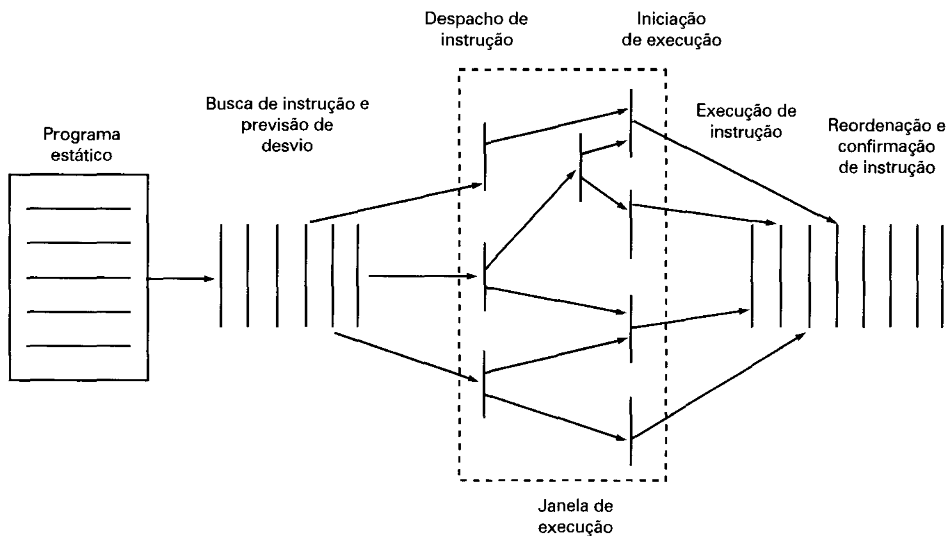
b. É possível agora efetuar algumas otimizações peculiares ao SPARC. A instrução `nop` depois da instrução `ble` pode ser substituída movendo outra instrução para essa posição de atraso e atribuindo valor 1 ao bit especial da instrução `ble` (expressa como `ble,a L1`). Mostre o programa obtido depois dessa mudança.

c. Existem agora duas instruções desnecessárias. Remova essas instruções e mostre o programa resultante.

capítulo

13

PARALELISMO NO NÍVEL DE INSTRUÇÕES E PROCESSADORES SUPERESCALARES



- Um processador superescalar é aquele no qual são usadas várias *pipelines* de instrução independentes. Cada *pipeline* tem diversos estágios, podendo manipular várias instruções a cada instante. O uso de várias *pipelines* introduz um novo nível de paralelismo, possibilitando processar diversos fluxos de instrução de cada vez. Um processador superescalar explora o que é conhecido como paralelismo no nível de instruções, que diz respeito ao nível em que instruções de um programa podem ser executadas em paralelo.
- Um processador superescalar busca tipicamente várias instruções de cada vez, e tenta encontrar instruções próximas que sejam independentes umas das outras e que podem, portanto, ser executadas em paralelo. Se os dados de entrada de uma instrução dependem da saída produzida por uma instrução precedente, a execução dessa instrução não pode ser completada antes ou ao mesmo tempo que a execução da instrução da qual ela depende. Uma vez que essas dependências de dados sejam identificadas, o processador pode executar e completar instruções em uma ordem diferente da que ocorre no código de máquina original.
- O processador pode eliminar dependências desnecessárias por meio do uso de registradores adicionais e de renomeação de referências a registradores no código original.
- Embora processadores RISC puros freqüentemente empreguem técnicas de atraso de execução de instruções de desvio para maximizar a utilização da *pipeline* de instruções, esse método não é muito adequado para uma máquina superescalar. Para melhorar a eficiência, a maioria das máquinas superescalares usa, em vez disso, métodos tradicionais de previsão de desvio.

Uma implementação superescalar de uma arquitetura de processador é uma implementação na qual instruções usuais — aritmética de números inteiros e de números de ponto flutuante, instruções de carga e armazenamento e instruções de desvio — podem ser iniciadas simultaneamente e executadas independentemente. Tais implementações suscitam diversas questões de projeto bastante complexas, relacionadas à *pipeline* de instruções.

O projeto superescalar apareceu logo depois dos projetos de arquitetura RISC. Embora as técnicas de arquitetura superescalar possam ser aplicadas mais diretamente a uma arquitetura RISC com um conjunto simplificado de instruções, a abordagem superescalar também pode ser usada em arquiteturas CISC.

Enquanto o período decorrido entre o início das pesquisas relativas à arquitetura RISC, com o IBM 801 e o Berkeley RISC I, até o lançamento de máquinas RISC comerciais tenha sido de sete ou oito anos, as primeiras máquinas superescalares tornaram-se disponíveis comercialmente apenas um ano ou dois após o aparecimento do termo *superescalar*. A abordagem superescalar tornou-se hoje o método padrão para implementar microprocessadores de alto desempenho.

Este capítulo começa com uma visão geral sobre a abordagem superescalar, contrastando-a com a abordagem de superpipeline. Em seguida, abordamos as principais questões de projeto associadas à implementação de máquinas superescalares. Depois, examinamos alguns exemplos importantes da arquitetura superescalar. Finalmente, abordamos a nova arquitetura IA-64, que pode ser considerada um projeto superescalar aprimorado.

13.1 VISÃO GERAL

O termo *superescalar*, usado originalmente em 1987 (Agerwala e Cocke, 1987), refere-se a máquinas projetadas para melhorar o desempenho da execução de instruções escalares. Esse nome evidencia o contraste entre o objetivo de projetos desse tipo e o de projetos de processadores vetoriais, discutido no Capítulo 16. Na maioria das aplicações, o maior volume de operações é sobre grandezas escalares. Por isso, a abordagem superescalar representa o próximo passo na evolução de processadores de propósito geral de alto desempenho.

A essência da abordagem superescalar é a habilidade de executar instruções independentemente, em diferentes *pipelines*. Esse conceito pode ser ainda mais explorado, permitindo que instruções sejam executadas em uma ordem diferente da ordem em que aparecem no programa. A Figura 13.1 ilustra a abordagem superescalar, em termos gerais. Existem várias unidades funcionais, cada uma das quais implementada como uma *pipeline*, provendo suporte à execução paralela de diversas instruções. Nesse exemplo, podem ser executadas, ao mesmo tempo, duas operações inteiras, duas operações de ponto flutuante e uma operação de memória (carga ou armazenamento).

Muitos pesquisadores têm investigado o uso de processadores do tipo superescalar, tendo concluído que é possível obter um certo grau de melhoria de desempenho. Alguns fatores de melhoria de desempenho (*speedup*) relatados são apresentados na Tabela 13.1. As diferenças entre esses resultados se devem tanto a diferenças de hardware entre as máquinas simuladas como a diferenças entre as aplicações usadas na simulação.

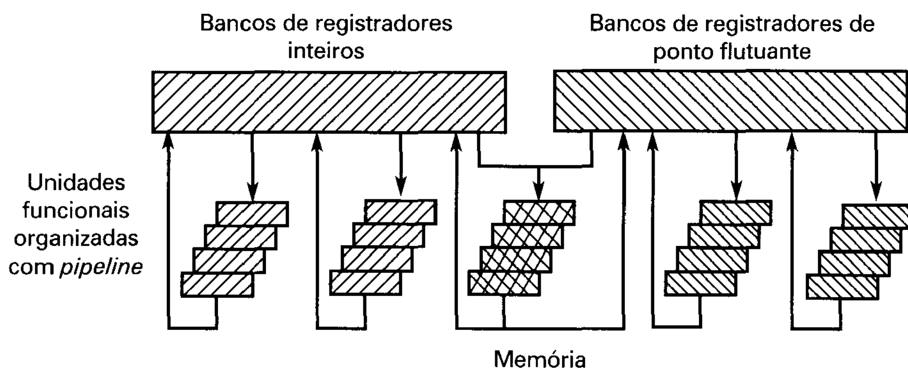


Figura 13.1 Organização geral de um processador com arquitetura superescalar (Comerford, 1995).

Tabela 13.1 Melhorias de desempenho relatadas para máquinas do tipo superescalar

Referência	Fator de aumento de desempenho
Tjaden e Flynn, 1970	1,8
Kuck, Muraoka e Chen, 1972	8
Weiss e Smith, 1984	1,58
Acosta, Kjelstrup e Torng, 1986	2,7
Sohi, 1990	1,8
Smith, Johnson e Horowitz, 1989	2,3
Jouppi, 1989b	2,2
Lee, 1991	7

Arquitetura superescalar *versus* arquitetura com *superpipeline*

Uma abordagem alternativa para obter maior desempenho é conhecida como superpipeline, termo primeiramente cunhado em 1988 (Jouppi, 1988). A técnica de superpipeline explora o fato de que muitos dos estágios de uma *pipeline* desempenham tarefas que requerem um tempo menor que a metade de um ciclo de relógio. Portanto, o uso de um relógio interno com o dobro de velocidade possibilitaria efetuar duas tarefas em cada ciclo do relógio externo. O processador MIPS R4000, visto anteriormente, constitui um exemplo dessa abordagem.

A Figura 13.2 compara essas duas abordagens. A parte superior do diagrama ilustra uma *pipeline* comum, usada como base para comparação. A *pipeline* básica emite uma instrução por ciclo de relógio e pode completar apenas um estágio da *pipeline* por ciclo de relógio. A *pipeline* tem quatro estágios: busca de instrução, decodificação da operação, execução da operação e escrita de resultado. Para maior clareza, o estágio da execução é representado de forma hachurada. Note que, embora sejam executadas diversas instruções simultaneamente, existe apenas uma instrução no estágio de execução a cada instante.

A parte seguinte do diagrama ilustra uma implementação com superpipeline, capaz de completar dois estágios da *pipeline* por ciclo de relógio. Uma forma alternativa de ver isso é considerar as operações realizadas em cada estágio como divididas em duas partes não so-

brepostas, cada qual podendo ser executada em meio ciclo de relógio. Uma implementação de uma superpipeline que se comporta desse modo é dita de grau 2. Finalmente, a parte inferior do diagrama mostra uma implementação de uma arquitetura superescalar, capaz de executar duas instâncias de cada estágio em paralelo. Implementações superescalares ou de superpipelines com grau superior também são possíveis.

Tanto a implementação de *superpipeline* como a superescalar, representadas na Figura 13.2, apresentam o mesmo número de instruções sendo executadas ao mesmo tempo, em estado estável. O processador com uma *superpipeline* gasta mais tempo do que o processador superescalar no início do programa e a cada instrução-alvo de um desvio tomado.

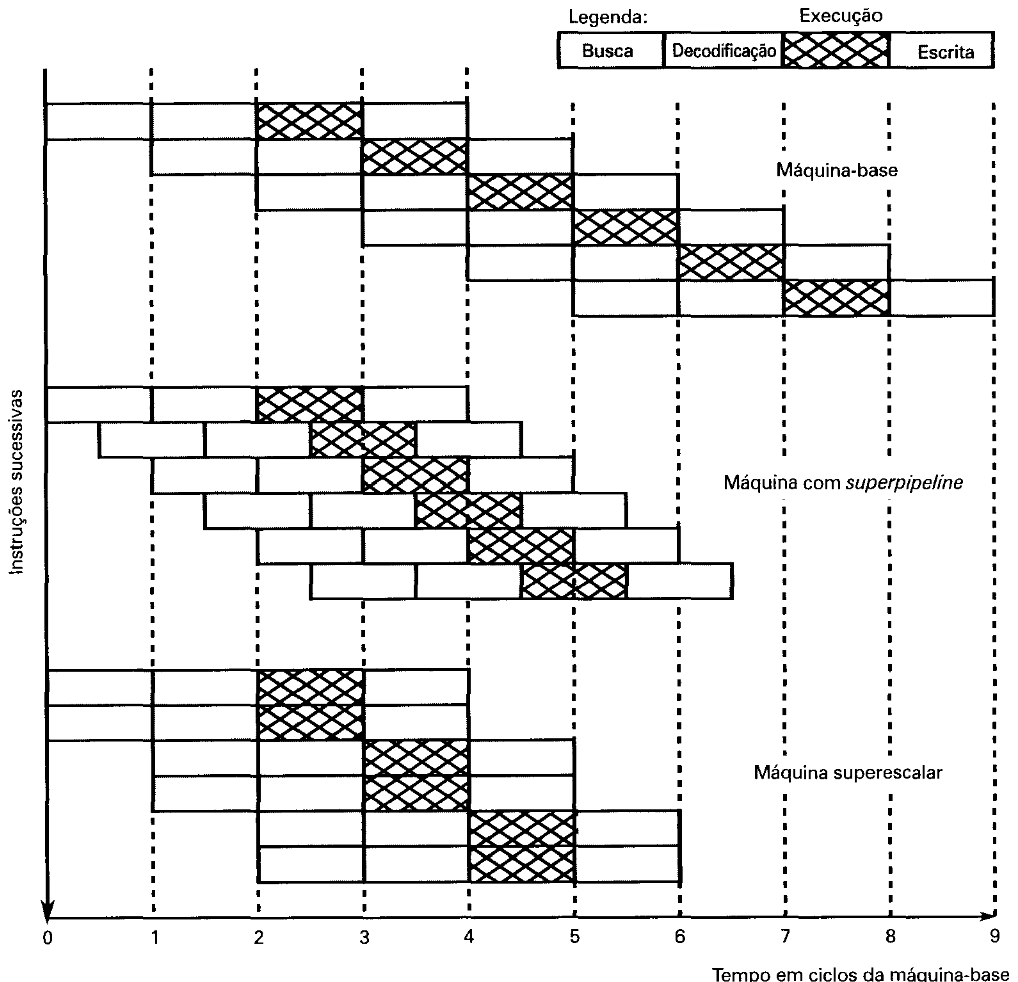


Figura 13.2 Comparação entre as abordagens superescalar e *superpipeline*.

Limitações

A abordagem superescalar depende da habilidade de executar várias instruções em paralelo. O termo **paralelismo no nível de instruções** diz respeito ao nível no qual as instruções

de um programa podem ser executadas em paralelo, em média. Para maximizar o paralelismo no nível de instruções, pode ser usada uma combinação de otimização baseada em compilador e técnicas de hardware. Antes de examinar as técnicas de projeto usadas em máquinas superescalares para aumentar o paralelismo no nível de instruções, precisamos perceber as limitações intrínsecas do paralelismo, com as quais o sistema deve lidar. Cinco limitações são relacionadas em Johnson (1991):

- Dependência de dados verdadeira
- Dependência de desvio
- Conflito de recurso
- Dependência de saída
- Antidependência

As três primeiras limitações são examinadas no restante desta seção. Uma discussão sobre as duas últimas é adiada até a introdução de alguns conceitos novos, na próxima seção.

Dependência de dados verdadeira

Considere a seguinte seqüência de instruções:

```
add     r1, r2      ;carregar registrador r1 com a soma dos conteúdos
                     de r1 e r2
move    r3, r1      ;carregar registrador r3 com o conteúdo de r1
```

A segunda instrução pode ser buscada e decodificada antecipadamente, mas não pode ser executada até que seja completada a execução da primeira instrução. A razão é que a segunda instrução requer dados produzidos pela primeira instrução. Essa situação é conhecida como dependência de dados verdadeira (também chamada dependência de fluxo ou dependência de escrita-leitura).

A Figura 13.3 ilustra essa dependência em uma máquina superescalar de grau 2. Caso não exista dependência de dados, duas instruções podem ser buscadas e executadas em paralelo. Se existir uma dependência de dados entre a primeira e a segunda instruções, então a execução da segunda instrução deve ser atrasada, pelo número de ciclos de relógio necessários para remover a dependência. De modo geral, uma instrução deve ser atrasada até que todos os seus dados de entrada tenham sido produzidos.

Em uma *pipeline* escalar simples, a seqüência de instruções mencionada acima não causaria atraso. Entretanto, considere o seguinte caso, em que um dos dados de entrada é carregado a partir da memória e não de um registrador:

```
load    r1, eff     ;carregar registrador r1 com o conteúdo da posição
                      de memória de endereço efetivo eff
move    r3, r1      ;carregar registrador r3 com o conteúdo de r1
```

Um processador RISC típico leva dois ou mais ciclos para transferir um valor da memória para um registrador, devido a atrasos no acesso à cache ou a uma memória externa à pastilha. Esse atraso pode ser eliminado fazendo com que o compilador reordene as instruções, de modo que uma ou mais instruções subseqüentes, que não dependam da leitura na memória, possam fluir por meio da *pipeline*. Esse esquema não é muito efetivo no caso de uma *pipeline* superescalar, pois as instruções independentes que são executadas durante o tempo

gasto para carregar um dado da memória provavelmente consomem apenas o primeiro ciclo da instrução de carga, deixando o processador ocioso até que essa instrução seja completada.

Dependência de desvios

Como discutido no Capítulo 11, a presença de desvios condicionais em uma seqüência de instruções complica a operação da *pipeline*. A instrução seguinte a um desvio condicional (tomado ou não) depende dessa instrução de desvio e não pode ser executada até que seja completada a execução da instrução de desvio. A Figura 13.3 ilustra o efeito de um desvio condicional em uma *pipeline* superescalar de grau 2.

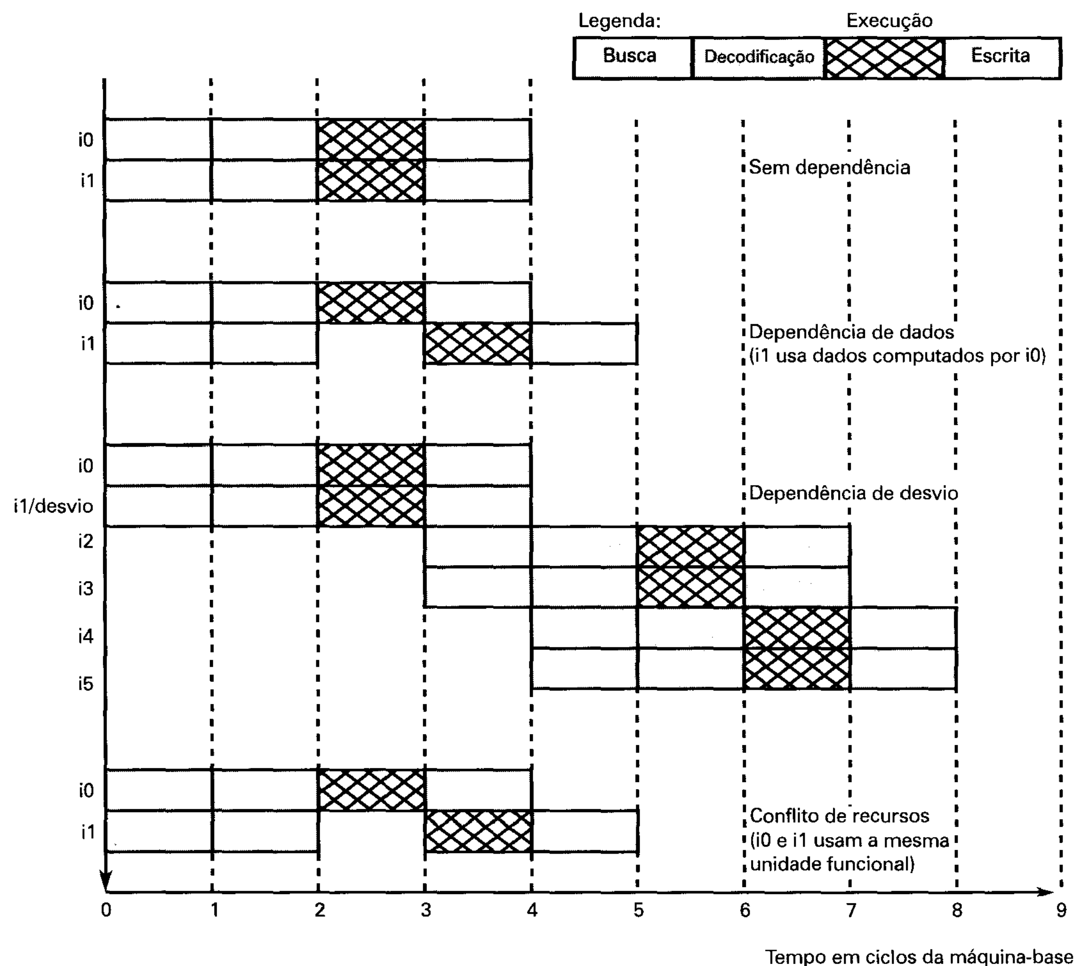


Figura 13.3 Efeito de dependências.

Como vimos anteriormente, esse tipo de dependência também afeta uma *pipeline* escalar. Novamente, a conseqüência desse tipo de dependência é mais severa em uma *pipeline* superescalar, porque o número de instruções perdidas em cada atraso é maior.

Se forem usadas instruções de tamanho variável, surge ainda outro tipo de dependência. Como o tamanho de uma instrução particular não é conhecido, uma instrução deve ser decodificada pelo menos parcialmente, antes que a instrução seguinte possa ser buscada. Isso impede a busca simultânea de instruções, requerida em uma *pipeline* superescalar. Essa é uma das razões pelas quais técnicas superescalares são mais diretamente aplicáveis a arquiteturas RISC ou do tipo RISC, que possuem instruções de tamanho fixo.

Conflito de recurso

Um conflito de recurso ocorre quando duas ou mais instruções competem, ao mesmo tempo, por um mesmo recurso. Exemplos de recursos incluem memórias, caches, barramentos, portas de bancos de registradores e unidades funcionais (por exemplo, o somador da ULA).

Em termos da *pipeline*, um conflito de recurso apresenta um comportamento semelhante ao de uma dependência de dados (Figura 13.3). Existem, entretanto, algumas diferenças. Por um lado, conflitos de recurso podem ser superados pela duplicação de recursos, enquanto uma dependência de dados não pode ser eliminada. Além disso, quando uma operação efetuada em uma dada unidade funcional consome muito tempo para ser completada, é possível minimizar os conflitos de uso dessa unidade por meio da sua implementação como uma *pipeline*.

13.2 QUESTÕES DE PROJETO

Paralelismo no nível de instruções e paralelismo de máquina

Jouppi e Wall (1989a) fazem distinção entre dois conceitos relacionados: paralelismo no nível de instruções e paralelismo de máquina. O **paralelismo no nível de instruções** existe quando as instruções de uma seqüência são independentes e podem, portanto, ser executadas em paralelo, por sobreposição.

Como exemplo do conceito de paralelismo no nível de instruções, considere os dois fragmentos de código a seguir (Jouppi, 1989b):

```
Load R1 ← R2              Add   R3 ← R3, "1"
Add R3 ← R3, "1"          Add   R4 ← R3, R2
Add R4 ← R4, R2           Store  [R4] ← R0
```

As três instruções à esquerda são independentes e, em tese, podem ser executadas paralelamente. As três instruções à direita, ao contrário, não podem ser executadas em paralelo, porque a segunda instrução usa o resultado da primeira, e a terceira instrução usa o resultado da segunda.

O grau de paralelismo no nível de instruções é determinado pela freqüência com que ocorrem no código dependências de dados verdadeiras e dependências de desvio. Esses fatores, por sua vez, são dependentes da aplicação e da arquitetura do conjunto de instruções. O paralelismo no nível de instruções é também determinado pelo que Jouppi e Wall (1989a) denominam latência de operação: o tempo até que o resultado de uma instrução esteja disponí-

vel para ser usado como operando da instrução subseqüente. Essa latência determina o atraso causado por uma dependência de dados ou uma dependência de desvio.

O **paralelismo de máquina** é uma medida da capacidade do processador em aproveitar o paralelismo no nível de instruções. O paralelismo de máquina é determinado pelo número de instruções que podem ser buscadas e executadas ao mesmo tempo (número de *pipelines* paralelas) e pela velocidade e eficácia dos mecanismos usados pelo processador para identificar instruções independentes.

Tanto o paralelismo no nível de instruções como o paralelismo de máquina são fatores importantes para melhoria do desempenho. Um programa pode não ter paralelismo no nível de instruções em grau suficiente para tirar total proveito do paralelismo de máquina. O uso de uma arquitetura com um conjunto de instruções de tamanho fixo, como em computadores RISC, aumenta o paralelismo no nível de instruções. Por outro lado, um paralelismo de máquina limitado restringe o desempenho, qualquer que seja a natureza do programa.

Política de iniciação de instruções

Como foi mencionado anteriormente, o paralelismo de máquina não é apenas uma questão de ter várias instâncias de cada estágio da *pipeline*. O processador deve também ser capaz de identificar paralelismo no nível de instruções e coordenar a busca, decodificação e execução de instruções, em paralelo. Johnson (1991) usa o termo **iniciação de instruções** (*instruction issue*) para referir-se ao processo de iniciar a execução de instruções nas unidades funcionais do processador, e o termo **política de iniciação de instrução** para referir-se ao protocolo usado para iniciar a execução de instruções.

Essencialmente, o processador procura examinar algumas instruções à frente do ponto de execução corrente, para localizar instruções que possam ser colocadas na *pipeline* para execução. Para isso, três tipos de ordenação de instruções são importantes:

- A ordem em que as instruções são buscadas
- A ordem em que as instruções são executadas
- A ordem em que as instruções atualizam o conteúdo de registradores e posições da memória

Quanto mais sofisticado o processador, menor é a sua limitação devido a essas ordenações. Para otimizar a utilização dos vários elementos da *pipeline,* o processador deve alterar a posição das instruções em uma ou mais dessas ordenações, com relação à ordem encontrada em uma execução estritamente seqüencial. A única restrição é a de que o resultado da execução de uma seqüência de instruções deve ser correto, isto é, igual ao obtido na execução seqüencial. O processador deve, portanto, se ajustar às várias dependências e conflitos discutidos anteriormente.

Em termos gerais, as políticas de iniciação de instruções em máquinas superescalares podem ser agrupadas nas seguintes categorias:

- Iniciação em ordem, com terminação em ordem
- Iniciação em ordem, com terminação fora de ordem
- Iniciação fora de ordem, com terminação fora de ordem

Iniciação em ordem com terminação em ordem

A política mais simples de iniciação de instruções consiste em iniciar as instruções na ordem exata em que elas seriam iniciadas em uma execução seqüencial, e escrever os resultados nessa mesma ordem (terminação de instruções em ordem). Nem mesmo *pipelines* escalares seguem essa política ingênua. Entretanto, é útil considerar essa política como base para a comparação com abordagens mais elaboradas.

A Figura 13.4a ilustra o uso dessa política. Considere uma *pipeline* superescalar, capaz de buscar e decodificar duas instruções ao mesmo tempo, tendo três unidades funcionais separadas (por exemplo, aritmética inteira, aritmética de ponto flutuante), e duas instâncias do estágio da *pipeline* que efetua escrita de resultados. O exemplo supõe as seguintes restrições, em um fragmento de código de seis instruções:

- I1 requer dois ciclos para ser executada
- I3 e I4 competem pela mesma unidade funcional
- I5 depende do valor produzido por I4
- I5 e I6 competem por uma unidade funcional

As instruções são buscadas, duas de cada vez, e passadas para a unidade de decodificação. Como as instruções são buscadas em pares, as duas próximas instruções devem esperar até que o par de estágios de decodificação da *pipeline* esteja vazio. Para garantir o término em ordem, quando existe um conflito por uma unidade funcional ou quando uma unidade funcional requer mais de um ciclo para gerar um resultado, a iniciação de instruções pára temporariamente.

Nesse exemplo, o tempo decorrido desde a decodificação da primeira instrução até a escrita dos últimos resultados é de oito ciclos.

Iniciação em ordem com terminação fora de ordem

A terminação fora de ordem é usada em processadores RISC escalares, para melhorar o desempenho de execução de instruções que requerem vários ciclos. A Figura 13.4b ilustra o uso dessa política em um processador superescalar. A execução da instrução I2 pode ser concluída antes que seja concluída a instrução anterior, I1. Isso permite que I3 seja completada mais cedo, resultando em uma economia de um ciclo.

Com a terminação fora de ordem pode haver, em qualquer instante, qualquer número de instruções no estágio de execução da *pipeline*, até o máximo grau de paralelismo de máquina existente ao longo de todas as unidades funcionais. A iniciação de instruções é temporariamente interrompida em caso de conflito por recurso, dependência de dados ou dependência de desvio.

Além dessas limitações, surge uma nova dependência, que foi denominada anteriormente **dependência de saída** (também chamada dependência de escrita-escrita). O seguinte fragmento de código ilustra essa dependência (op representa uma operação qualquer):

```
I1:   R3  ← R3 op R5
I2:   R4  ← R3  +  1
I3:   R3  ← R5  +  1
I4:   R7  ← R3 op R4
```

Decodificação		Execução			Escrita		Ciclo
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

(a) Iniciação em ordem e terminação em ordem

Decodificação		Execução			Escrita		Ciclo
I1	I1						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

(b) Iniciação em ordem e terminação fora de ordem

Decodificação		Janela	Execução			Escrita		Ciclo
I1	I2							1
I3	I4	I1,I2	I1	I2				2
I5	I6	I3,I4	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

(c) Iniciação fora de ordem e terminação fora de ordem

Figura 13.4 Políticas de iniciação e terminação de instruções em máquinas superescalares.

A instrução I2 não pode ser executada antes da instrução I1, porque precisa que o resultado produzido por I1 esteja armazenado no registrador R3; isso constitui um exemplo de dependência de dados verdadeira, como descrita na Seção 13.1. Semelhantemente, a instrução I4 deve esperar por I3, porque usa um resultado produzido por essa instrução. E qual é a relação entre I1 e I3? Nesse caso, não existem dependências de dados, tal como definimos anteriormente. Entretanto, se I3 for completada antes de I1, o valor de R3 usado na execução de I4 estará errado. Portanto, I3 deve ser completada depois de I1, para que os valores de saída produzidos sejam corretos. Para garantir isso, a iniciação da terceira instrução deve ser atrasada, caso o seu resultado possa ser mais tarde sobrescrito por uma instrução mais antiga, cuja execução leve mais tempo para ser completada.

A terminação fora de ordem requer uma lógica de iniciação de instruções mais complexa que no caso da terminação em ordem. Além disso, é mais difícil lidar com interrupções e exceções. Quando ocorre uma interrupção, a execução da instrução é suspensa no ponto atual,

para ser retomada mais tarde. O processador deve assegurar que a retomada da execução leve em conta o fato de que, no momento da interrupção, já podem ter sido completadas instruções subseqüentes à instrução que causou a interrupção.

Iniciação fora de ordem com terminação fora de ordem

Com a iniciação em ordem, o processador decodifica instruções apenas até o ponto em que ocorre uma dependência ou conflito. Nenhuma instrução adicional é decodificada até que o conflito seja resolvido. Como resultado, o processador não pode examinar instruções adiante do ponto de conflito, para tentar identificar instruções subseqüentes que sejam independentes das instruções já presentes na *pipeline* e que poderiam, portanto, ser introduzidas na *pipeline* produzindo resultado útil.

Para possibilitar a iniciação de instruções fora de ordem, é necessário desvincular os estágios de decodificação e de execução da *pipeline*. Isso é feito usando uma área de armazenamento temporário, chamada de **janela de instruções**. Com essa organização, depois que o processador termina a decodificação de uma instrução, ela é colocada na janela de instruções. O processador pode continuar a busca e a decodificação de novas instruções, desde que a janela de instruções não esteja cheia. Quando uma unidade funcional se torna disponível no estágio de execução, pode ser iniciada uma instrução da janela de instruções para o estágio de execução. Qualquer instrução pode ser iniciada, desde que (1) necessite da unidade funcional particular que está disponível e (2) nenhum conflito ou dependência bloqueie essa instrução.

O resultado dessa organização é que o processador tem capacidade de examinar instruções à frente, podendo identificar instruções independentes que possam ser trazidas para o estágio de execução. As instruções são iniciadas da janela de instrução sem necessariamente respeitar a ordem em que originalmente ocorrem no programa. Como antes, a única restrição é que a execução do programa se comporte corretamente.

A Figura 13.4c ilustra essa política. Em cada ciclo, são buscadas duas instruções para o estágio de decodificação. Em cada ciclo, duas instruções são movidas do estágio de decodificação para a janela de instruções, desde que exista espaço disponível na janela de instruções. Nesse exemplo, é possível iniciar a instrução I6 antes da instrução I5 (lembre-se que I5 depende de I4, mas I6 não). Portanto, é economizado um ciclo, tanto no estágio de execução como no estágio de escrita de resposta. A economia obtida ao longo de toda a *pipeline*, em comparação com a Figura 13.4b, é de um ciclo.

A janela de instruções é mostrada na Figura 13.4c para ilustrar sua utilização. Entretanto, essa janela não é um estágio adicional da *pipeline*. A presença de uma instrução na janela significa simplesmente que o processador tem informação suficiente sobre essa instrução para decidir quando ela pode ser iniciada.

A política de iniciação fora de ordem com terminação fora de ordem está sujeita às mesmas restrições descritas anteriormente. Uma instrução não pode ser iniciada se introduz uma dependência ou conflito. A diferença é que um maior número de instruções está disponível para iniciação, o que reduz a probabilidade de que um estágio da *pipeline* tenha de ser suspenso. Além das dependências vistas anteriormente, surge uma nova dependência, à qual nos referimos anteriormente como **antidependência** (também chamada dependência de leitura-escrita). O fragmento de código considerado anteriormente ilustra essa dependência:

```
I1:  R3  ← R3 op R5
I2:  R4  ← R3  + 1
I3:  R3  ← R5  + 1
I4:  R7  ← R3 op R4
```

A instrução I3 não pode ser completada antes que tenha sido iniciada a execução da instrução I2 e seus operandos tenham sido buscados. Isso porque I3 atualiza o registrador R3, que constitui um operando fonte para a instrução I2. O termo *antidependência* é usado porque essa restrição é semelhante a uma dependência de dados verdadeira, mas é invertida: em vez de a primeira instrução produzir um valor que é usado na segunda instrução, a segunda instrução destrói um valor que é usado pela primeira.

Renomeação de registradores

Quando são permitidas a iniciação de instruções fora de ordem e/ou a terminação fora de ordem, existe a possibilidade de ocorrência de dependências de saída e antidependências. Essas dependências são diferentes das dependências de dados verdadeiras e dos conflitos de recurso, que refletem o fluxo de dados através do programa e a seqüência de execução de instruções. Dependências de saída e antidependências, por outro lado, surgem porque os valores contidos nos registradores podem não mais refletir a seqüência de valores ditada pelo fluxo do programa.

Quando as instruções são iniciadas em seqüência e terminadas em seqüência, é possível especificar o conteúdo de cada registrador, em cada ponto da execução. Quando são usadas técnicas de iniciação ou de terminação fora de ordem, não é possível determinar os valores de todos os registradores em cada instante, com base apenas na seqüência de instruções ditada pelo programa. De fato, os valores competem pelo uso dos registradores, e o processador deve resolver esses conflitos que ocasionalmente causam a suspensão de um estágio da *pipeline*.

Dependências de saída e antidependências são exemplos de conflito de armazenamento. Várias instruções competem pelo uso dos mesmos registradores, gerando restrições que diminuem o desempenho da *pipeline*. O problema se torna mais agudo quando são usadas técnicas de otimização de registradores (como discutido no Capítulo 12), uma vez que essas técnicas de compilação buscam maximizar a utilização de registradores, maximizando assim o número de conflitos de armazenamento.

Um método para lidar com esses tipos de conflito de armazenamento é baseado em uma técnica tradicional de resolução de conflito de recurso: a duplicação de recursos. Nesse contexto, a técnica é chamada de **renomeação de registradores**. Essencialmente, os registradores são alocados dinamicamente pelo hardware do processador, sendo associados aos valores requeridos pelas instruções a cada instante de tempo. Quando é produzido um novo valor que deve ser armazenado em um registrador (por exemplo, quando a instrução que está sendo executada tem um registrador como operando de destino), um novo registrador é criado para esse valor. Instruções subseqüentes que tenham como operando fonte o valor armazenado nesse registrador devem seguir um processo de renomeação de registradores: referências a registradores existentes nessas instruções são revisadas, passando a referir-se aos registradores que contêm os valores requeridos. Portanto, referências originais a um mesmo registrador em diversas instruções diferentes podem ser convertidas, de fato, em referências reais a registradores distintos, caso os valores pretendidos sejam diferentes.

Vejamos como a renomeação de registradores pode ser usada no fragmento de código que examinamos anteriormente:

$$
\begin{aligned}
I1 &: R3_b \leftarrow R3_a \text{ op } R5_a \\
I2 &: R4_b \leftarrow R3_b + 1 \\
I3 &: R3_c \leftarrow R5_a + 1 \\
I4 &: R7_b \leftarrow R3_c \text{ op } R4_b
\end{aligned}
$$

Uma referência a registrador representada sem o subscrito indica uma referência a registrador lógico, encontrada na instrução. Uma referência a registrador com o subscrito indica o registrador do hardware alocado para conter o novo valor. Quando é feita alocação de um novo registrador para corresponder a um determinado registrador lógico, referências a esse registrador lógico como operando fonte em instruções subseqüentes são convertidas em referências para o registrador de hardware associado a ele mais recentemente (recente em termos da seqüência de instruções do programa).

Nesse exemplo, a criação do registrador $R3_c$ na instrução I3 evita a ocorrência de antidependência na segunda instrução e de dependência de saída na primeira instrução; além disso, ele não impede que o valor correto seja usado pela instrução I4. O resultado é que I3 pode ser iniciada imediatamente; se a renomeação de registradores não fosse usada, I3 não poderia ser iniciada antes que a primeira instrução fosse terminada e a segunda instrução fosse iniciada.

Paralelismo de máquina

Examinamos anteriormente três técnicas de hardware que podem ser usadas em um processador superescalar para melhorar o desempenho: duplicação de recursos, iniciação de instruções fora de ordem e renomeação de registradores. Um estudo relatado em (Smith, Johnson e Horowitz (1989) ilumina a relação entre essas técnicas. Esse estudo utiliza a simulação para modelar uma máquina com as características do processador MIPS R2000, aumentado com várias características superescalares. Foram simuladas diversas seqüências de instruções de programa diferentes.

A Figura 13.5 mostra os resultados. Em cada um dos gráficos, o eixo vertical corresponde ao aumento médio da velocidade de execução (*speedup*) da máquina superescalar em relação à máquina escalar. O eixo horizontal mostra os resultados obtidos para quatro organizações alternativas do processador. A máquina-base não tem nenhuma das unidades funcionais duplicada, mas pode iniciar instruções fora de ordem. A segunda configuração duplica a unidade de carga/armazenamento que acessa uma memória cache de dados. A terceira configuração duplica a ULA e a quarta configuração duplica a unidade de carga/amazenamento e a ULA. Em cada gráfico, são mostrados os resultados obtidos com janelas de instruções de tamanho igual a 8, 16 e 32 instruções, o que determina o número de instruções à frente que o processador é capaz de examinar. A diferença entre os dois gráficos é que, no segundo, é usada renomeação de registradores. Isso equivale a dizer que o primeiro gráfico reflete uma máquina que é limitada por todas as dependências, enquanto o segundo gráfico corresponde a uma máquina limitada apenas por dependências reais (não introduzidas devido ao uso de iniciação e execução de instruções fora de ordem).

A partir da combinação dos dois gráficos, podem ser tiradas conclusões importantes. A primeira é que provavelmente não vale a pena adicionar unidades funcionais sem usar renomeação de registradores. Embora haja uma ligeira melhora de desempenho, ela não justifica

tre instruções. Por isso, as máquinas superescalares retornaram às técnicas de previsão de desvio usadas antes das máquinas RISC. Algumas máquinas superescalares, como o PowerPC 601, utilizam a técnica simples de previsão estática de desvio. Processadores mais sofisticados, tais como o PowerPC 620 e o Pentium II, usam previsão dinâmica de desvio, baseada em análises de histórico de desvios.

Execução superescalar

Podemos agora dar uma visão geral do processo de execução de programas em uma máquina superescalar: isso é ilustrado na Figura 13.6. O programa a ser executado consiste de uma seqüência linear de instruções. Esse é o programa estático, tal como escrito pelo programador ou gerado pelo compilador. O processo de busca de instruções, que inclui previsão de desvio, é usado para construir um fluxo dinâmico de instruções. O processador examina esse fluxo para determinar dependências, podendo remover dependências artificiais. As instruções são então despachadas para uma janela de execução. Nessa janela, as instruções não formam mais um fluxo seqüencial, mas são ordenadas de acordo com suas dependências de dados verdadeiras. O processador efetua o estágio de execução de cada instrução em uma ordem determinada pelas dependências de dados verdadeiras. Finalmente, as instruções são recolocadas na ordem seqüencial de execução e seus resultados são registrados.

O passo final mencionado no parágrafo anterior é conhecido como *confirmação* de instrução. Esse passo é necessário pela seguinte razão. Devido ao uso de múltiplas *pipelines* em paralelo, as instruções podem ser completadas em uma ordem diferente daquela em que seriam concluídas em uma execução seqüencial do programa estático. Além disso, o uso de previsão de desvio e de execução especulativa faz com que algumas instruções possam ser completadas, e então, seus resultados serem descartados porque o ramo de desvio que elas representam não foi tomado. Portanto, a memória e os registradores visíveis para o programa não podem ser atualizados imediatamente, quando as instruções são completadas. Os resultados de cada instrução devem ser mantidos em algum tipo de área de armazenamento temporário, que possa ser usada pelas instruções dependentes, e então serem armazenados de forma permanente apenas quando for determinado que a instrução realmente seria executada no modelo de execução seqüencial.

Implementação superescalar

Com base na discussão feita até aqui, podemos fazer alguns comentários gerais sobre o hardware requerido em um processador projetado com abordagem superescalar. Smith e Sohi (1995) listam os seguintes elementos básicos:

- Estratégias de busca de instrução, que buscam múltiplas instruções simultaneamente, freqüentemente prevendo o resultado de desvios condicionais. Essas funções requerem o uso de múltiplos estágios de busca e de decodificação na *pipeline*, e de lógica de previsão de desvios.
- Lógica para determinar dependências de dados verdadeiras envolvendo os valores armazenados em registradores, e mecanismos para transferir esses valores para os pontos onde são necessários durante a execução.
- Mecanismos para iniciar múltiplas instruções em paralelo.

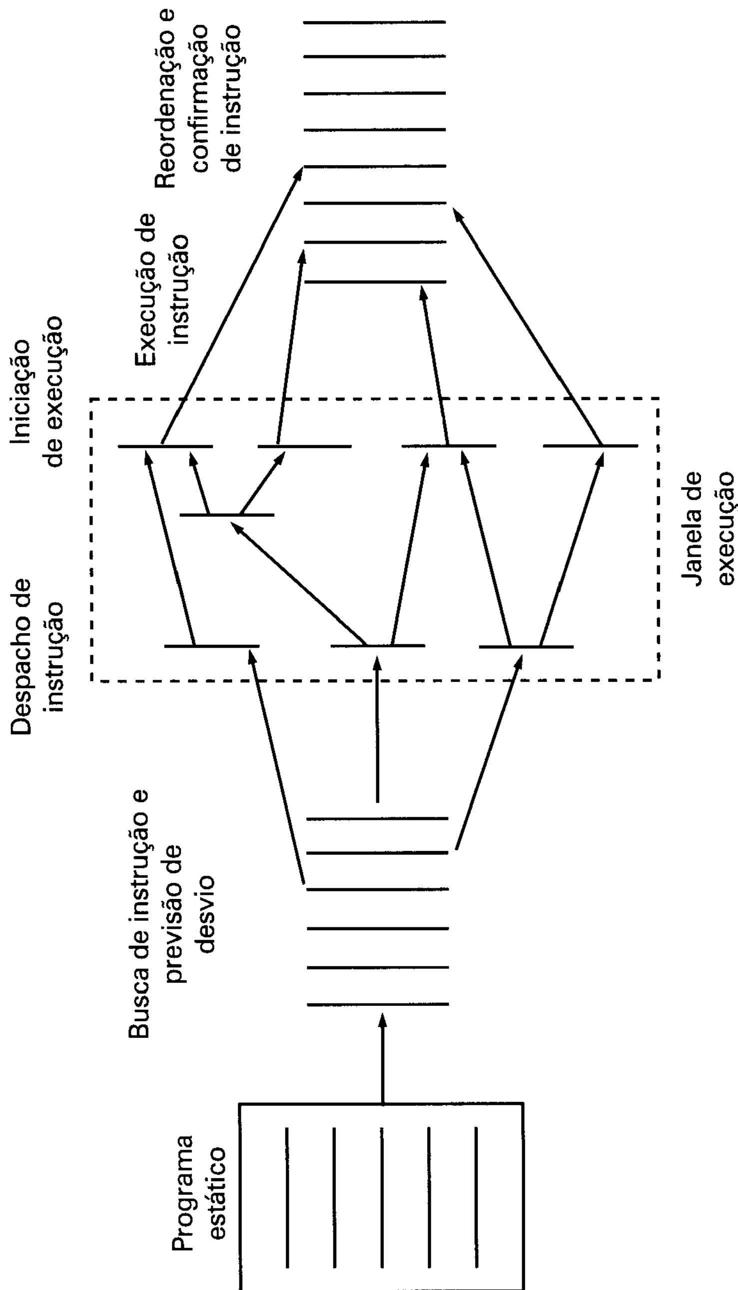


Figura 13.6 Representação conceitual de processamento superescalar (Smith e Sohi, 1995).

- Recursos para execução paralela de múltiplas instruções, incluindo múltiplas unidades funcionais paralelas e hierarquia de memória capaz de servir simultaneamente múltiplas referências à memória.
- Mecanismos para confirmar resultados do processamento na ordem correta.

13.3 PENTIUM II

Embora o conceito de projeto superescalar geralmente seja associado com a arquitetura RISC, os mesmos princípios podem ser aplicados a máquinas CISC. O exemplo mais notável disso é talvez o Pentium II. É interessante notar a evolução dos conceitos de arquitetura superescalar na linha Intel. O 80486 foi uma máquina CISC tradicional, sem elementos superescalares. O Pentium tinha um modesto componente superescalar, consistindo do uso de duas unidades de execução inteira. O Pentium Pro introduziu um projeto superescalar mais maduro. O Pentium II tem essencialmente a mesma organização superescalar do Pentium Pro, com o acréscimo de unidades de execução MMX.

Um diagrama de blocos geral do Pentium II foi mostrado na Figura 4.23. Os componentes essenciais da organização superescalar são a unidade de busca e decodificação de instruções, a unidade de despacho e execução e a unidade de confirmação. Cada uma dessas unidades será examinada separadamente, depois da breve visão geral apresentada a seguir.

A operação do Pentium II pode ser resumida do seguinte modo:

1. O processador busca instruções na memória, na ordem em que ocorrem no programa estático.
2. Cada instrução é traduzida em uma ou mais instruções RISC de tamanho fixo, conhecidas como microoperações.
3. O processador executa as microoperações em uma organização de *pipeline* superescalar, de forma que as microoperações podem ser executadas fora de ordem.
4. O processador submete os resultados da execução de cada microoperação para o conjunto de registradores do processador na ordem em que ocorreriam no fluxo original do programa.

A arquitetura do Pentium II consiste, de fato, de um revestimento exterior CISC e um núcleo interno RISC. As microoperações RISC internas passam através de uma *pipeline* com pelo menos 11 estágios (Figura 13.7); em alguns casos, a microoperação requer múltiplos estágios de execução, resultando em uma *pipeline* ainda maior. Isso contrasta com a *pipeline* de cinco estágios (Figura 11.18) usada nos processadores Intel x86 e no Pentium.

Unidade de busca e decodificação de instrução

A Figura 13.8 é uma visão simplificada da unidade de busca e decodificação do Pentium II. A operação de busca consiste em uma *pipeline* de três estágios. Primeiro, o estágio IFU1 busca instruções da cache de instruções, uma linha (32 bytes) de cada vez. A unidade Next_IP fornece o endereço da próxima instrução a ser buscada e a linha da cache que contém essa instrução é trazida para uma área de armazenamento temporário de instruções do estágio IFU1. A determinação da próxima instrução não é calculada simplesmente incrementando o apontador de instrução, porque pode haver um desvio ou uma interrupção pendente, que atualizam o apontador de instruções com um endereço de instrução diferente.

Em seguida, o conteúdo da área de armazenamento temporário de instruções do estágio IFU1 é passado para o estágio IFU2, 16 bytes de cada vez. O estágio IFU2 desempenha duas operações em paralelo. Ele examina os 16 bytes de instruções para determinar os limites de cada instrução; essa operação é necessária devido ao tamanho variável de instruções do Pentium. Se qualquer das instruções for um desvio, essa unidade passa o endereço de memória correspondente para a lógica de previsão dinâmica de desvio. O IFU2 passa então o bloco de

16 bytes para o estágio IFU3, que é responsável por alinhar as instruções para serem apresentadas ao decodificador adequado.

Para entender a operação do estágio IFU3, precisamos descrever o primeiro estágio da unidade de decodificação de instrução, ID1. Esse estágio é capaz de manipular três instruções de máquina Pentium II em paralelo. O estágio ID1 traduz cada instrução de máquina em uma seqüência de uma a quatro microoperações, cada uma das quais é uma instrução RISC de 118 bits. Note que, na maioria das máquinas RISC puras, as instruções têm tamanho de apenas 32 bits. O tamanho maior das microoperações é necessário para acomodar as operações mais complexas do Pentium. Entretanto, as microoperações são mais fáceis de gerenciar que as instruções originais das quais elas derivam. O estágio ID1 contém três decodificadores. O primeiro deles manipula instruções do Pentium que são traduzidas em até quatro microoperações. O segundo e terceiro decodificadores manipulam apenas instruções simples, que são mapeadas em uma única microoperação; essas incluem instruções simples de transferência de dados entre registradores e instruções de carga. Para acomodar-se à estrutura do estágio ID1, o estágio IFU3 efetua, se necessário, um deslocamento circular do conteúdo da sua área de armazenamento temporário de instruções, de 16 bytes, para que a primeira instrução nessa área seja uma instrução complexa e as duas instruções seguintes sejam instruções simples. Se as três instruções contidas nessa área forem instruções simples, a rotação não será necessária. Se mais de uma instrução for complexa, então as instruções devem ser passadas ao estágio ID1 em etapas, de forma que nenhuma instrução complexa seja apresentada ao segundo e ao terceiro decodificadores.

Poucas instruções requerem mais que quatro microoperações. Essas instruções são transferidas para o seqüenciador de instrução de microcódigo (MIS), que é uma ROM de microcódigos que contém as séries de microoperações (cinco ou mais) associadas a cada instrução de máquina complexa. Por exemplo, uma instrução de manipulação de cadeia de caracteres pode ser traduzida em uma grande seqüência repetitiva de microoperações (até mesmo com centenas de microoperações). O MIS é, portanto, uma unidade microprogramada, como discutido na Parte 4.

A saída do estágio ID1 ou do MIS alimenta o segundo estágio da unidade de codificação, ID2, em blocos de até seis microoperações de cada vez. O estágio ID2 enfileira as microoperações na ordem original em que ocorrem no programa. Nesse ponto, existe uma segunda oportunidade para previsão de desvio. Qualquer microoperação de desvio é apresentada para a unidade de previsão estática de desvio, que passa sua saída para a unidade de previsão dinâmica de desvio. A previsão de desvios é descrita no decorrer desta seção.

As microoperações enfileiradas no estágio ID2 passam por uma fase de renomeação de registradores, no que é conhecido por unidade de alocação de registradores (RAT). O RAT converte referências aos 16 registradores da arquitetura (registradores EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP, além dos 8 registradores de ponto flutuante) para referências a registradores de um conjunto de 40 registradores físicos. Esse estágio remove falsas dependências, causadas pelo número limitado de registradores da arquitetura, preservando apenas as dependências de dados verdadeiras (leitura após escrita). O RAT então passa as microoperações revisadas para o estágio de reordenação (ROB).

IFU1	IFU2	IFU3	ID1	ID2	RAT	ROB	DIS	EX	RET1	RET2

IFU = Unidade de Busca de Instrução
ID = Unidade de Decodificação de Instrução
RAT = Unidade de Alocação de Registradores
ROB = Área de Reordenação de Instruções
DIS = Unidade de Despacho de Instrução
EX = Estágio de Execução
RET = Unidade de Confirmação

Figura 13.7 A *pipeline* do Pentium II.

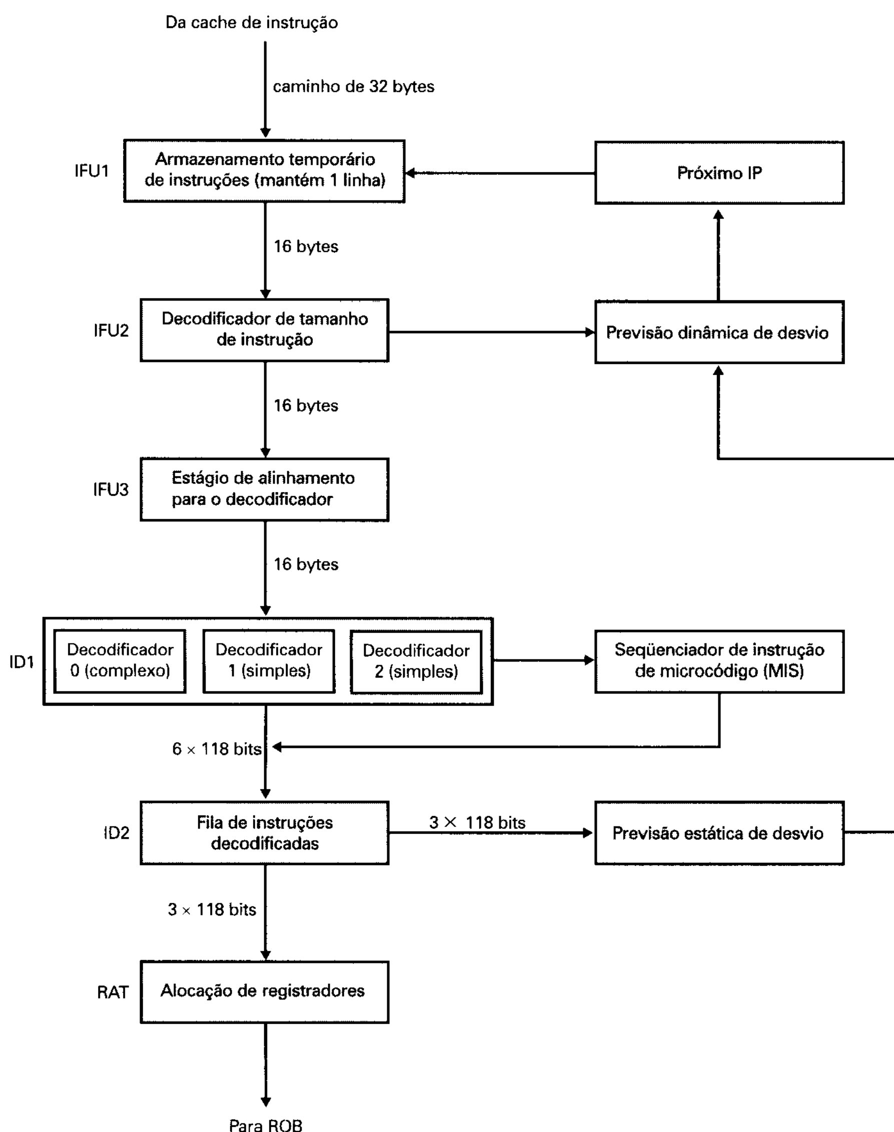


Figura 13.8 Unidade de busca e decodificação do Pentium II.

O fluxo direto de microoperações através da unidade de despacho/execução é perturbado em caso de previsão incorreta de desvio. Quando se verifica que uma previsão de desvio foi incorreta, podem já existir microoperações em vários estágios de processamento, que devem ser removidas da *pipeline*. Isso é responsabilidade da unidade de execução de desvio (JEU). Quando uma operação de desvio é executada, o resultado do desvio é comparado ao resultado previsto pelo hardware de previsão de desvio. Se esses resultados não coincidem, a unidade de execução de desvio altera o estado de todas as microoperações posteriores ao desvio, de modo que sejam removidas da área de armazenamento temporário de microoperações. O destino apropriado do desvio é então fornecido ao hardware de previsão de desvio, que reinicia toda a *pipeline* a partir do novo endereço-alvo.

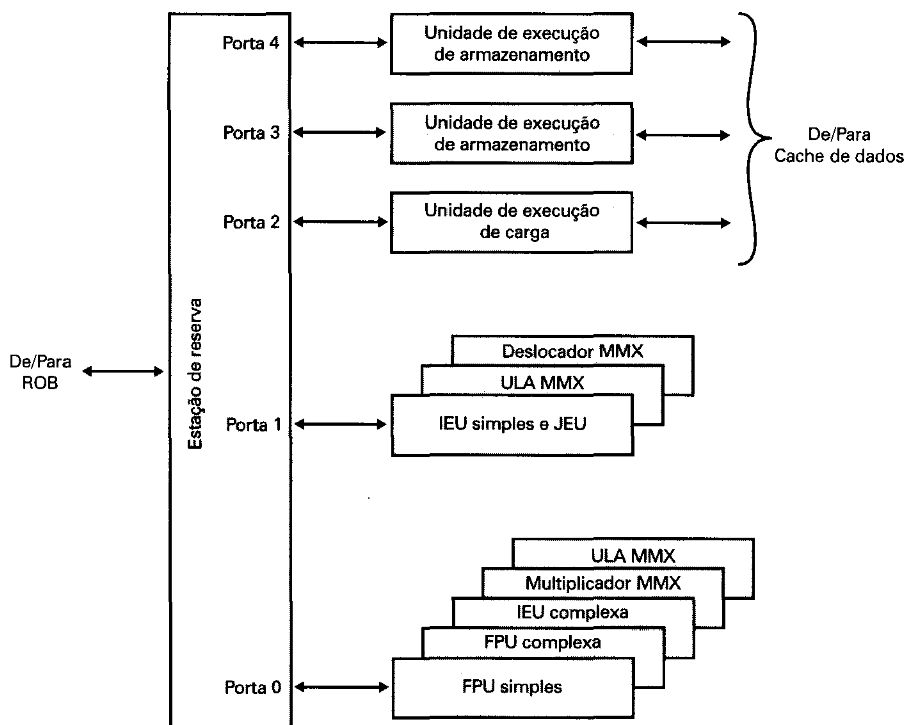


Figura 13.9 Unidade de despacho/execução do Pentium II.

Unidade de confirmação

A unidade de confirmação (RU — *retire unit*) funciona fora da área de reordenação (ROB) e confirma resultados da execução de instruções. A RU deve levar em conta previsões de desvio incorretas e microoperações que já tenham sido executadas, mas que sucedem desvios que ainda não foram validados. Quando se determina que uma microoperação foi executada e que ela não está sujeita a ser retirada da *pipeline* devido a uma previsão de desvio incorreta, essa microoperação é marcada como pronta para confirmação. Quando a instrução Pentium anterior já tiver sido confirmada, e todas as microoperações correspondentes à instrução seguinte estiverem marcadas como prontas para a confirmação, a RU atualiza os registradores da arquitetura afetados por essa instrução e apaga suas microoperações da ROB.

Previsão de desvio

O Pentium II usa uma estratégia dinâmica de previsão de desvio, baseada no histórico de execuções recentes de instruções de desvio. Uma tabela de endereços-alvo de desvios (BTB) armazena informação sobre instruções de desvio executadas mais recentemente. Quando uma instrução de desvio é encontrada no fluxo de instrução, essa tabela é consultada. Se existir uma entrada correspondente na BTB, a unidade de execução será guiada pela informação de histórico contida nessa entrada para prever se o desvio será tomado ou não. Se for previsto que o desvio será tomado, o endereço de destino associado a essa entrada será usado para buscar antecipadamente a instrução-alvo do desvio.

Uma vez executada uma instrução de desvio, seu histórico é atualizado de modo que reflita o resultado da instrução. Se a instrução não estiver presente na BTB, seu endereço será carregado em uma entrada dessa tabela; se necessário, uma entrada mais antiga será apagada.

A descrição apresentada nos dois parágrafos anteriores corresponde, em termos gerais, à estratégia de previsão de desvio usada no Pentium, no Pentium Pro e no Pentium II. Entretanto, no caso do Pentium, é usado um esquema de histórico relativamente simples com 2 bits. O Pentium Pro e o Pentium II possuem *pipelines* muito maiores (com 11 ou mais estágios, comparados aos 5 estágios do Pentium) e, portanto, a perda em caso de previsão incorreta de desvio é muito maior. Por isso, o Pentium Pro e o Pentium II usam um esquema de previsão de desvio mais elaborado, com maior número de bits de histórico, para reduzir a taxa de previsão incorreta.

A BTB do Pentium II é organizada como uma cache associativa por conjuntos de quatro linhas, com um total de 512 linhas. Cada entrada usa como rótulo o endereço do desvio, e inclui o endereço de destino da última vez que o desvio foi tomado e um campo de histórico de 4 bits. O uso de quatro bits de histórico contrasta com os 2 bits usados originalmente no Pentium, assim como na maioria dos processadores superescalares. Usando quatro bits, o mecanismo do Pentium II leva em conta um histórico mais longo para prever desvios. O algoritmo usado é conhecido como algoritmo de Yeh (Yeh e Patt, 1991). Os pesquisadores que propuseram esse algoritmo mostraram que ele fornece uma redução significativa do número de previsões de desvio incorretas, se comparado com algoritmos que usam apenas 2 bits de histórico (Evers et al, 1998).

Desvios condicionais que não tenham histórico na BTB são previstos por meio de um algoritmo de previsão estática, de acordo com as seguintes regras:

- No caso de desvio em que o endereço-alvo não é relativo ao IP, a previsão é de tomar o desvio, se este for um retorno. Caso contrário, a previsão é a de não tomar o desvio.
- Para desvios condicionais relativos ao IP para trás, ou seja, para endereços anteriores ao da instrução corrente, a previsão é de que o desvio será tomado. Essa regra reflete o comportamento típico de laços de repetição.
- No caso de desvios condicionais relativos a IP para a frente, ou seja, para instruções posteriores à instrução corrente, a previsão é de que o desvio não será tomado.

13.4 PowerPC

A arquitetura do PowerPC descende diretamente das arquiteturas IBM 801, RT PC e RS/6000, esta última também referenciada como uma implementação da arquitetura POWER. Todas elas são máquinas RISC; mas a RS/6000 foi a primeira da série a exibir características superescalares. A primeira implementação da arquitetura PowerPC, o processador 601, tem um projeto superescalar bastante semelhante ao da arquitetura RS/6000. Os modelos PowerPC subseqüentes levaram mais adiante o conceito superescalar. Nesta seção, enfocamos o PowerPC 601, que fornece um bom exemplo de projeto superescalar baseado em arquitetura RISC. No final da seção, abordamos brevemente o processador 620.

PowerPC 601

A Figura 13.10 dá uma visão geral da organização do processador PowerPC 601. Assim como outras máquinas superescalares, o PowerPC 601 é dividido em unidades funcionais independentes, para aumentar a oportunidade de execuções sobrepostas. Em particular, o núcleo do PowerPC 601 consiste de três unidades de execução independentes, organizadas como *pipeline*: a unidade de número inteiro, a unidade de ponto flutuante e a unidade de processamento de desvio. Juntas, essas unidades podem executar três instruções ao mesmo tempo, resultando em um projeto superescalar de grau 3.

A Figura 13.11 mostra uma visão lógica da arquitetura PowerPC 601, enfatizando o fluxo de instruções entre as unidades funcionais. A unidade de busca pode obter antecipadamente da cache até oito instruções ao mesmo tempo. A unidade de cache provê uma cache combinada de instruções e dados e é responsável por suprir instruções para as outras unidades e por suprir dados para os registradores. A lógica de arbitração da cache envia para a cache o endereço de acesso de maior prioridade.

Unidade de despacho

A unidade de despacho pega instruções da cache e as carrega em uma fila de despacho, que pode conter oito instruções de cada vez. Essa unidade processa o fluxo de instruções de modo que alimente com um fluxo constante de instruções as unidades de número inteiro, de número de ponto flutuante e de processamento de desvio. A metade superior da fila atua simplesmente como uma área de armazenamento temporário, para manter instruções até que elas sejam movidas para a parte inferior. Seu propósito é assegurar que a unidade de despacho não se atrase esperando por instruções da cache. As instruções armazenadas na metade inferior são despachadas de acordo com o seguinte esquema:

- **Unidade de processamento de desvio**: manipula todas as instruções de desvio condicional. A primeira das instruções de desvio contidas na metade inferior da fila de despacho é enviada para a unidade de processamento de desvio, caso essa unidade possa aceitá-la.
- **Unidade de ponto flutuante**: manipula todas as instruções de ponto flutuante. A primeira instrução desse tipo contida na metade inferior da fila de despacho é enviada para a unidade de ponto flutuante, se a *pipeline* de instrução dessa unidade não estiver cheia.

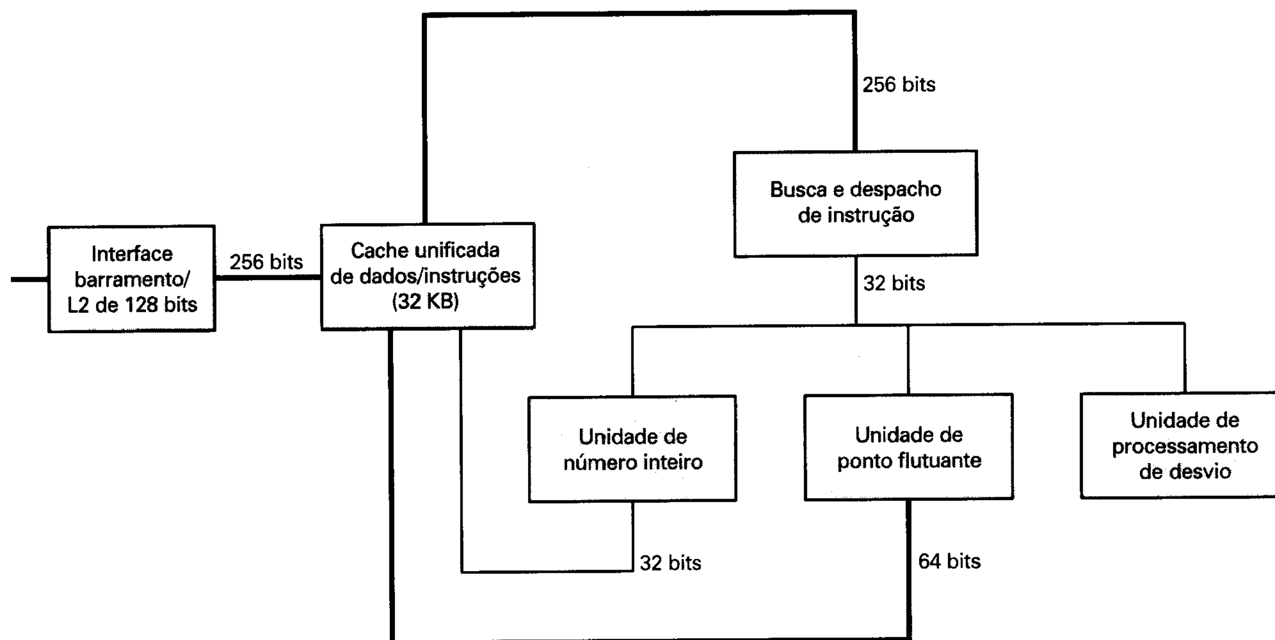


Figura 13.10 Diagrama de bloco do PowerPC 601.

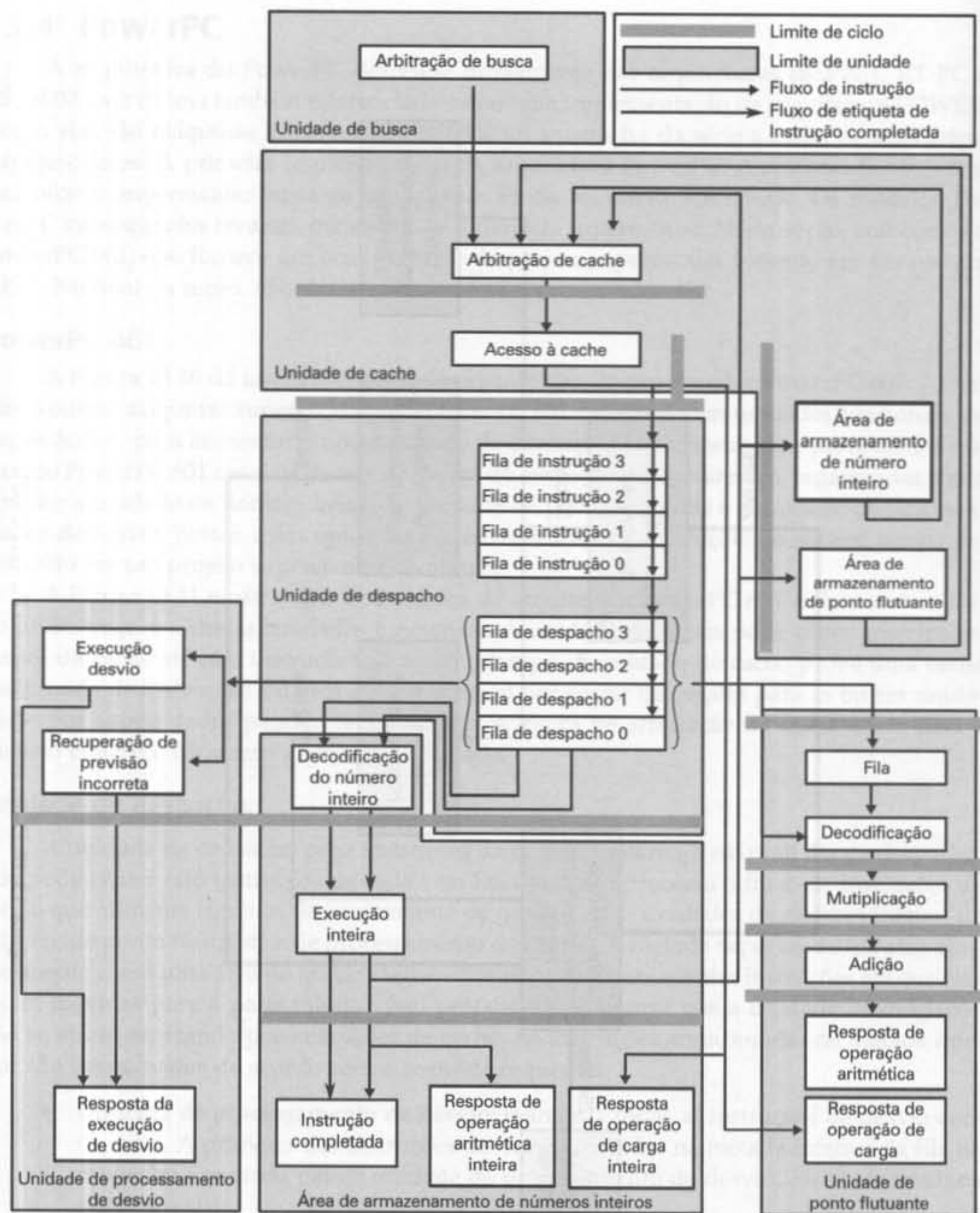


Figura 13.11 Estrutura de *pipeline* do PowerPC 601 (Potter et al, 1994).

- **Unidade de número inteiro**: manipula instruções de número inteiro, instruções de carga/armazenamento entre bancos de registradores e a cache e instruções de comparação de números inteiros. Uma instrução de número inteiro apenas é emitida depois de ter sido filtrada para a metade inferior da fila de despacho.

Permitindo que instruções de desvio e instruções de ponto flutuante sejam emitidas fora de ordem, faz com que o processador mova instruções por meio da fila de despacho o mais rapidamente possível, buscando manter cheias as *pipelines* das unidades de execução de desvios e de instruções de ponto flutuante.

A unidade de despacho contém também a lógica para calcular o endereço para busca antecipada de instrução. Ela busca instruções seqüencialmente, até que uma instrução de desvio seja movida para a metade inferior da fila de despacho. Quando a unidade de processamento de desvio processa uma instrução, ela pode atualizar o endereço de busca antecipada, para que as instruções seguintes sejam buscadas a partir do novo endereço e incluídas na fila de despacho.

Pipelines de Instrução

A Figura 13.12 ilustra as *pipelines* de instrução das várias unidades. O ciclo de busca é comum a todas as instruções e ocorre antes que a instrução seja despachada para uma unidade particular. O segundo ciclo começa com o despacho de uma instrução para uma unidade particular. Isso se sobrepõe a outras atividades dentro da unidade. Durante cada ciclo de relógio, a unidade de despacho examina as quatro entradas localizadas mais no fundo da fila de despacho de instruções e despacha até três instruções.

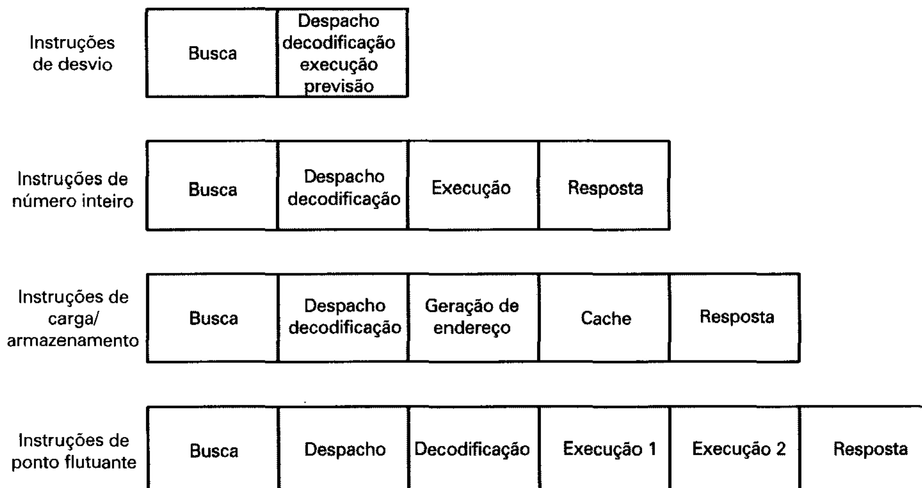


Figura 13.12 *Pipeline* de PowerPC 601.

No caso de instruções de desvio, o segundo ciclo envolve a decodificação e a execução das instruções, assim como a previsão de desvios. Essa última atividade é discutida na próxima subseção.

A unidade de número inteiro lida com instruções que causam operações de carga/armazenamento na memória (incluindo carga/armazenamento de ponto flutuante), transferências de dados entre registradores ou operações da ULA. No caso de carga/armazenamento na memória, existe um ciclo de geração de endereço, seguido de envio do endereço resultante para a cache e, caso necessário, de um ciclo de escrita do resultado. Nas demais instruções a cache não é envolvida, existindo apenas um ciclo de execução seguido por uma escrita em registrador.

Instruções de ponto flutuante seguem uma *pipeline* semelhante, mas existem dois ciclos de execução, refletindo a complexidade das operações de ponto flutuante.

Vale a pena fazer algumas outras observações. O registrador de condição contém oito campos de código independentes, cada qual com 4 bits. Isso permite reter múltiplos códigos de condição, o que reduz a dependência mútua entre instruções. Por exemplo, o compilador pode transformar a seqüência:

```
comparação
desvio
comparação
desvio
•
•
•
```

na seqüência:

```
comparação
comparação
•
•
•
desvio
desvio
•
•
•
```

Como cada unidade funcional pode enviar seus códigos de condição para diferentes campos do registrador de condição, é possível evitar o bloqueio mútuo de instruções causado pelo compartilhamento de códigos de condição.

O uso dos registradores de salvamento e restauração de estado da máquina (SRRs) no processador de desvio possibilita manipular interrupções simples e interrupções de software sem envolver lógica das outras unidades funcionais. Portanto, serviços simples do sistema operacional podem ser desempenhados rapidamente, sem qualquer manipulação complicada de informações de estado ou sincronização entre unidades funcionais.

Como o 601 pode emitir instruções de desvio e de ponto flutuante fora de ordem, são necessários controles adicionais para assegurar a execução na ordem apropriada. Quando existe uma dependência (por exemplo, quando uma instrução requer um operando que ainda tem de ser computado por uma instrução anterior), a *pipeline* da unidade correspondente pára.

Processamento de desvio

A capacidade de otimizar o uso da *pipeline* constitui o ponto-chave para o alto desempenho de uma máquina RISC ou superescalar. Tipicamente, o elemento mais crítico do projeto é a forma como os desvios são manipulados. No PowerPC, o processamento de desvios é responsabilidade da unidade de desvio. A unidade é projetada de forma que, em muitos casos, um desvio não tem efeito sobre o ritmo de execução das outras unidades; desvios desse tipo

são conhecidos como desvios de zero-ciclos. Para obter um desvio de zero-ciclos são empregadas as seguintes estratégias:

1. Usa-se uma lógica para examinar a fila de despacho de instruções e identificar desvios. O endereço-alvo de um desvio é gerado quando ele aparece na metade inferior da fila e não existe nenhum desvio anterior cuja execução está pendente.
2. O processador de desvio procura determinar o resultado de desvios condicionais. Esse resultado pode ser determinado se o código de condição já tiver sido estabelecido. Assim que uma instrução de desvio é encontrada, o processador de desvio determina que:
   a. O desvio será tomado, para desvios incondicionais e para desvios condicionais cujo código de condição é conhecido e indica um desvio.
   b. O desvio não será tomado, para desvios condicionais cujo código de condição é conhecido e indica a não-ocorrência de desvio.
   c. O resultado não pode ser ainda determinado. Nesse caso, a lógica de previsão de desvio prevê que o desvio será tomado, se o desvio for para trás (típico de laços de repetição), e que não será tomado, se o desvio for para a frente. As instruções posteriores à instrução de desvio no programa original são passadas para as unidades de execução de forma condicional. Uma vez que o valor do código de condição seja produzido na unidade de execução, a unidade de desvio cancela as instruções presentes na *pipeline* e prossegue a partir do endereço-alvo buscado, caso se determine que o desvio foi tomado; caso contrário, ela sinaliza que as instruções condicionais devem ser executadas. Um bit especial no código de instruções de desvio pode ser usado pelo compilador para inverter esse comportamento padrão.

A incorporação de uma estratégia de previsão de desvio baseada em histórico foi rejeitada pelos projetistas, por acreditarem que o benefício a ser obtido seria mínimo.

Como exemplo do efeito da previsão de desvio, considere o programa da Figura 13.13 e suponhe que o processador de desvio prevê que a instrução de desvio condicional não é tomada (padrão no caso de desvio para a frente). A Figura 13.14a mostra o efeito sobre a *pipeline,* caso o desvio de fato não seja tomado. No primeiro ciclo, a fila de despacho é carregada com oito instruções. As primeiras seis instruções são instruções de números inteiros e são despachadas, uma por ciclo, para a unidade de número inteiro. A instrução de desvio condicional não pode ser despachada até que chegue à metade inferior da fila de despacho, o que acontece no ciclo 5. A unidade de desvio prevê que o desvio não será tomado; e, assim, a próxima instrução na seqüência é despachada condicionalmente (indicada por D′). O desvio não pode ser resolvido até que seja executada a instrução de comparação no ciclo 8. Nesse momento, o processador de desvio confirma que sua previsão foi correta e a execução continua. Não há nenhum atraso e a *pipeline* permanece cheia.

Note que não foi buscada nenhuma instrução durante os ciclos 4 a 8. Isso porque, durante esses ciclos, a cache permanece ocupada com o estágio de acesso à cache das cinco instruções de carga. Mesmo assim, o fluxo de instrução não é atrasado, porque a fila de despacho pode conter oito instruções.

A Figura 13.14b mostra o resultado caso a previsão tenha sido incorreta e o desvio seja tomado. Nessa hipótese, as três instruções que estão iniciando a partir do rótulo IF devem ser descarregadas da *pipeline,* e a busca de instruções é retomada a partir do ELSE. Como resul-

tado, o estágio de execução da *pipeline* de instruções de número inteiro fica ocioso durante os ciclos 9 e 10, ocorrendo uma perda de dois ciclos devido à previsão incorreta de desvio.

PowerPC 620

O PowerPC 620 é a primeira implementação de 64 bits da arquitetura PowerPC. A Figura 4.25 mostra um diagrama de bloco geral desse processador, que é igual ao do processador G3, mais recente. Uma característica dessa implementação é que ela inclui seis unidades de execução independentes:

- Unidade de instrução
- Três unidades de números inteiros
- Unidade de carga/armazenamento
- Unidade de ponto flutuante

Essa organização possibilita ao processador despachar até quatro instruções simultaneamente para as três unidades de números inteiros e para a unidade de ponto flutuante.

O PowerPC 620 emprega uma estratégia de previsão de desvio de alto desempenho, que envolve uma lógica de previsão, renomeação de registradores e estações de reserva dentro das unidades de execução. Quando uma instrução é buscada, são alocados registradores temporários para conter o seu resultado. Com o uso de renomeação de registradores, o processador pode executar instruções de forma *especulativa*, com base na previsão de desvio; se a previsão vier a ser incorreta, os resultados das instruções executadas de forma especulativa serão eliminados, sem danificar o banco de registradores. Uma vez que uma previsão de desvio seja confirmada, os resultados temporários poderão ser armazenados permanentemente.

Cada unidade tem duas ou mais estações de reserva, para armazenar instruções já despachadas que devem esperar pelo resultado de outras instruções. Com essa característica, essas instruções podem ser retiradas da unidade de instrução, possibilitando que ela continue a despachar instruções para outras unidades de execução.

O PowerPC 620 pode executar de forma especulativa até quatro instruções de desvio não resolvidas (apenas uma pode ser executada no PowerPC 601). A previsão de desvio é baseada no uso de uma tabela de histórico de desvio com 2048 entradas. Simulações executadas pelos projetistas do PowerPC mostraram que a taxa de sucesso na previsão de desvios é de 90% (Thompson e Ryan, 1994).

```
if (a > 0)
        a = a + b + c + d + e;
else
        a = a - b - c - d - e;
```

(a) Código em C

```
                            # r1 aponta para a,
                            # r1 + 4 aponta para b,
                            # r1 + 8 aponta para c,
                            # r1 + 12 aponta para d,
                            # r1 + 16 aponta para e.
   lwz     r8=a(r1)         # carregar a
   lwz     r12=b(r1, 4)     # carregar b
```

```
    lwz     r9=c(r1, 8)              # carregar c
    lwz     r10=d(r1,12)             # carregar d
    lwz     r11=e(r1,16)             # carregar e
    cmpi    cr0=r8,0                 # comparar com valor imediato
    bc      ELSE,cr0/gt=false        # desviar se o bit é falso
IF:
    add     r12=r8, r12              # adicionar
    add     r12=r12, r9              # adicionar
    add     r12=r12, r10             # adicionar
    add     r4=r12,r11               # adicionar
    stw     a(r1)=r4                 # armazenar
    b       OUT                      # desvio incondicional
ELSE:
    subf r12=r12, r8                 # subtrair
    subf r12=r9, r12                 # subtrair
    subf r12=r10, r12                # subtrair
    subf r4=r12, r11                 # subtrair
    stw  a(r1)-r4                    # armazenar
OUT:
```

(b) Código em linguagem de montagem

Figura 13.13 Exemplo de código com desvio condicional (Weiss e Smith, 1994).

```
                                      1 2 3 4 5 6 7 8  9 10 11 12 13 14 15 16
        lwz    r8=a(r1)               F D E C W
        lwz    r12=b(r1, 4)           F • D E C W
        lwz    r9=c(r1, 8)            F • • D E C W
        lwz    r10=d(r1,12)           F • • • D E C W
        lwz    r11=e(r1, 16)          F • • • • D E C  W
        cmpi   cr0=r8,0               F • • • • • D E
        bc     ELSE,cr0/gt=false      F • • • S
IF:     add    r12=r8, r12            F • • • • • • D' E  W
        add    r12=r12, r9                F • • • • •  D  E  W
        add    r12=r12, r10               F • • • • •  •  D  E  W
        add    r4=r12,r11                              F  •  D  E  W
        stw    a(r1)=r4                                F  •  •  D  E  C
        b      OUT
ELSE:   subf   r12=r8, r12
        subf   r12=r12, r9
        subf   r12=r12, r10
        subf   r4=r12, r11
        stw    a(r1)=r4
OUT:
```

(a) Previsão correta: desvio não tomado

Figura 13.14 Previsão de desvio: não tomado (Weiss e Smith, 1994). *(continua)*

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	lwz	r8=a(r1)	F	D	E	C	W											
	lwz	r12=b(r1, 4)	F	•	D	E	C	W										
	lwz	r9=c(r1, 8)	F	•	•	D	E	C	W									
	lwz	r10=d(r1,12)	F	•	•	•	D	E	C	W								
	lwz	r11=e(r1, 16)	F	•	•	•	•	D	E	C	W							
	cmpi	cr0=r8,0	F	•	•	•	•	•	D	E								
	bc	ELSE,cr0/gt=false	F	•	•	•	S											
IF:	add	r12=r8, r12	F	•	•	•	•	•	•	D'								
	add	r12=r12, r9			F	•	•	•	•	•								
	add	r12=r12, r10			F	•	•	•	•	•								
	add	r4=r12,r11																
	stw	a(r1)=r4																
	b	OUT																
ELSE:	subf	r12=r8, r12									F	D	E	W				
	subf	r12=r12, r9									F	•	D	E	W			
	subf	r12=r12, r10									F	•	•	D	E	W		
	subf	r4=r12, r11									F	•	•	•	D	E	W	
	stw	a(r1)=r4									F	•	•	•	•	D	E	C
OUT:																		

(b) Previsão incorreta: desvio tomado

F = busca
D = despacho/decodificação
E = execução/endereçamento
C = acesso à cache
W = escrita da resposta
S = despacho de instrução

Figura 13.14 Previsão de desvio: não tomado (Weiss e Smith, 1994). *(continuação)*

13.5 MIPS R10000

O MIPS R10000, que evoluiu a partir do MIPS R4000 e usa o mesmo conjunto de instruções, é uma implementação bastante limpa e direta de princípios de projeto superescalar. A Figura 13.15 mostra sua estrutura completa.

O primeiro estágio do processamento de instrução do R10000 é um estágio de pré-decodificação de instrução. A **pré-decodificação** é uma função encontrada em diversas máquinas superescalares, incluindo o PowerPC 620 e o UltraSparc. Ela auxilia na adequação do fluxo de instruções à demanda de instruções para as *pipelines* de execução paralela. Em qualquer máquina superescalar, a unidade de decodificação e envio de instrução, que inclui uma análise de dependências e de desvios, constitui um caminho crítico para se obter um alto fluxo de instruções. Isto porque é responsável por sustentar um fluxo de instruções suficiente para as múltiplas unidades de execução. Portanto, essa unidade representa um potencial gargalo no processador. Para acelerar a busca e decodificação de instruções, diversos processadores superescalares efetuam uma decodificação parcial das instruções, assim que elas são carregadas da cache secundária externa para a cache interna de instruções. A pré-decodificação classifica as instruções recebidas e anexa a cada uma um certo número de bits de decodificação, para simplificar o restante do processamento. No caso do R10000, são anexados a cada instrução quatro bits de decodificação, que indicam a unidade de execução para a qual essa instrução será finalmente despachada.

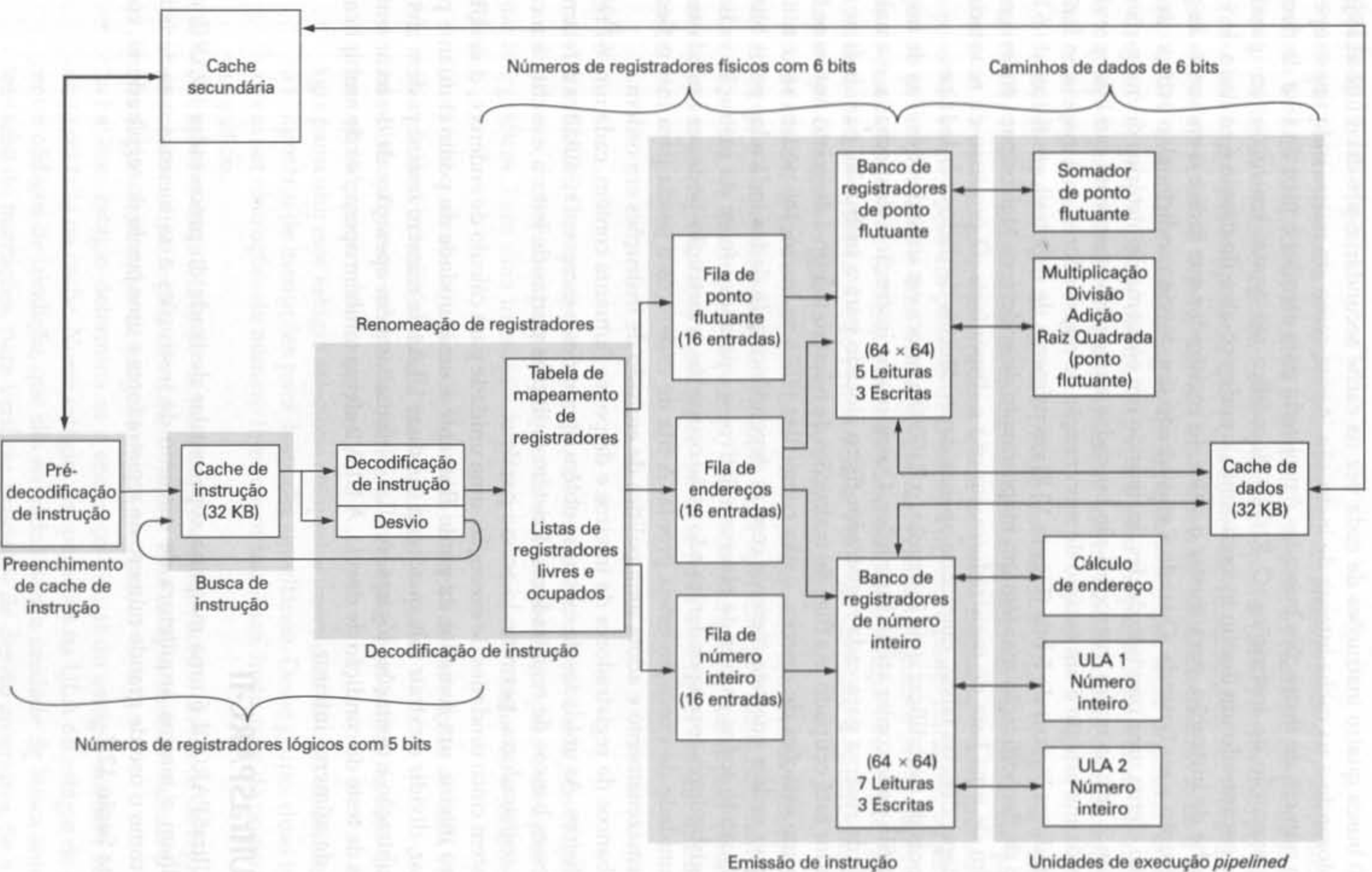


Figura 13.15 Estrutura do MIPS R10000.

O R10000 busca quatro instruções de cada vez na cache secundária, pré-decodificando cada uma e colocando-a na cache interna de instrução. Assim como em outras máquinas superescalares, o número de instruções buscadas é projetado para atender o pico da taxa de decodificação e execução de instruções. O R10000 decodifica até quatro instruções em paralelo. Quando é encontrado um desvio incondicional, o endereço-alvo do desvio é enviado de volta para a cache de instrução, para ajustar o valor do registrador que indica o endereço da próxima instrução a ser buscada. Quando é encontrado um desvio condicional, o estágio de decodificação efetua uma previsão de desvio, usando um esquema de histórico de instruções de desvio com 2 bits; as novas instruções são buscadas especulativamente no caminho previsto.

O próximo estágio é um estágio de renomeação de registradores, que mapeia endereços de registradores lógicos de 5 bits (Figura 12.8) em números de registradores físicos de 6 bits. O estágio de decodificação mantém um mapeamento de endereços lógicos para endereços físicos, além de uma lista de registradores em uso e disponíveis. O processo de renomeação remove dependências falsas, mantendo apenas dependências de dados verdadeiras.

Depois de decodificar uma instrução, o R10000 a coloca em uma das três filas de instrução. Cada fila pode conter até 16 instruções. Quando uma instrução é despachada, é atualizado um bit de reserva para cada registrador físico alocado para a instrução, para indicar que o registrador está ocupado. As filas de instruções de número inteiro e de ponto flutuante funcionam como estações de reserva, e não como filas FIFO; as instruções podem ser emitidas em qualquer ordem que seja coerente com as dependências de dados indicadas pelos bits de reserva. Quando todos os bits de reserva relativos a operandos fonte da instrução indicam que os registradores correspondentes não estão ocupados, a instrução pode ser emitida assim que sua unidade de execução esteja pronta. A fila de endereços é usada para instruções de carga e armazenamento e adota uma política de emissão de instruções em ordem.

Os bancos de registradores de inteiros e de ponto flutuante contêm, cada um, 64 registradores físicos. As unidades de execução obtêm operandos e escrevem resultados diretamente sobre esses bancos de registradores. Existem múltiplas portas de leitura e escrita em cada banco de registradores, permitindo acesso paralelo.

Existem cinco unidades de execução: uma unidade para cálculo de endereço, duas ULAs de número inteiro, um somador de ponto flutuante e uma unidade de ponto flutuante para multiplicar, dividir e extrair raiz quadrada. As duas ULAs de número inteiro podem efetuar adição, subtração e operações lógicas. A ULA 1 efetua também operações de deslocamento e operações de teste de condição de desvio. A ULA 2 efetua também operações de multiplicação e divisão de números inteiros.

13.6 UltraSPARC-II

O UltraSPARC II é uma máquina superescalar derivada do processador SPARC; elas compartilham a mesma arquitetura de conjunto de instruções e as mesmas características RISC, tal como o uso de grande número de registradores e uma janela de registradores, como descrito na Seção 12.7.

Organização interna

A Figura 13.16 mostra a estrutura interna de uma versão recente do processador, conhecida como UltraSPARC Ili. Uma cache externa L2 alimenta as caches L1 separadas de instruções e de dados. A unidade de busca antecipada e despacho (PDU) busca instruções em uma área de armazenamento temporário, que pode conter até 12 instruções. A PDU é também responsável por previsão de desvio, usando uma tabela de histórico de desvio de 2 bits. Finalmente, a PDU inclui um módulo lógico de agrupamento, que tenta organizar as instruções que chegam em grupos de até quatro instruções para despacho simultâneo. De cada grupo, duas instruções podem ser instruções de número inteiro e duas podem ser instruções de ponto flutuante ou gráficas.

A unidade de execução de número inteiro contém duas ULAs completas e oito janelas de registradores. Duas instruções de números inteiros podem ser executadas em paralelo. A unidade de ponto flutuante contém duas ULAs de ponto flutuante e uma unidade gráfica. Isso possibilita a execução em paralelo de duas instruções de ponto flutuante, ou de uma instrução de ponto flutuante e uma instrução gráfica. A unidade gráfica provê suporte ao conjunto de instruções visuais (VIS), que estende o conjunto de instruções do SPARC. Semelhante ao conjunto de instruções MMX do Pentium II, o VIS é um conjunto de instruções especializado para processamento gráfico (Tremblay e outros, 1996).

A unidade de carga e armazenamento (LSU) gera o endereço virtual para todas as operações de carga e armazenamento, executando uma operação por ciclo. Entretanto, em conjunto com a fila de instruções de carga, ela pode comprimir múltiplas operações de armazenamento sobre endereços confinados em uma faixa de 8 bytes em um único acesso à cache L2.

Pipeline

O UltraSPARC II emprega uma *pipeline* de instruções de nove estágios (Figura 13.17), que se divide em duas partes para instruções de número inteiro e para instruções de ponto flutuante e gráficas. Para uma instrução de número inteiro, são usados os seguintes estágios da *pipeline*:

- **Busca**: até quatro instruções são buscadas na cache de instruções de cada vez. A previsão de desvio também é feita nesse estágio.
- **Decodificação**: esse estágio decodifica as instruções e as coloca na área de armazenamento temporário de instruções.
- **Agrupamento**: esse estágio seleciona até quatro instruções da área de armazenamento temporário de instruções para despache simultâneo. Desse grupo, duas instruções podem ser instruções de número inteiro e outras duas, instruções de ponto flutuante ou gráficas.
- **Execução**: as duas ULAs de número inteiro efetuam acesso ao banco de registradores e executam as instruções de número inteiro. Esse estágio também calcula o endereço virtual para instruções de carga/armazenamento.
- **Cache**: esse estágio determina se o endereço virtual do estágio anterior resulta em acerto ou falta na cache. Nesse estágio, as operações na ULA do estágio de execução geram códigos de condição, que são enviados para a unidade de busca antecipada e despacho de instruções, para verificar previsões de desvio anteriores. Se a previsão for incorreta, as instruções anteriores serão eliminadas da *pipeline*.

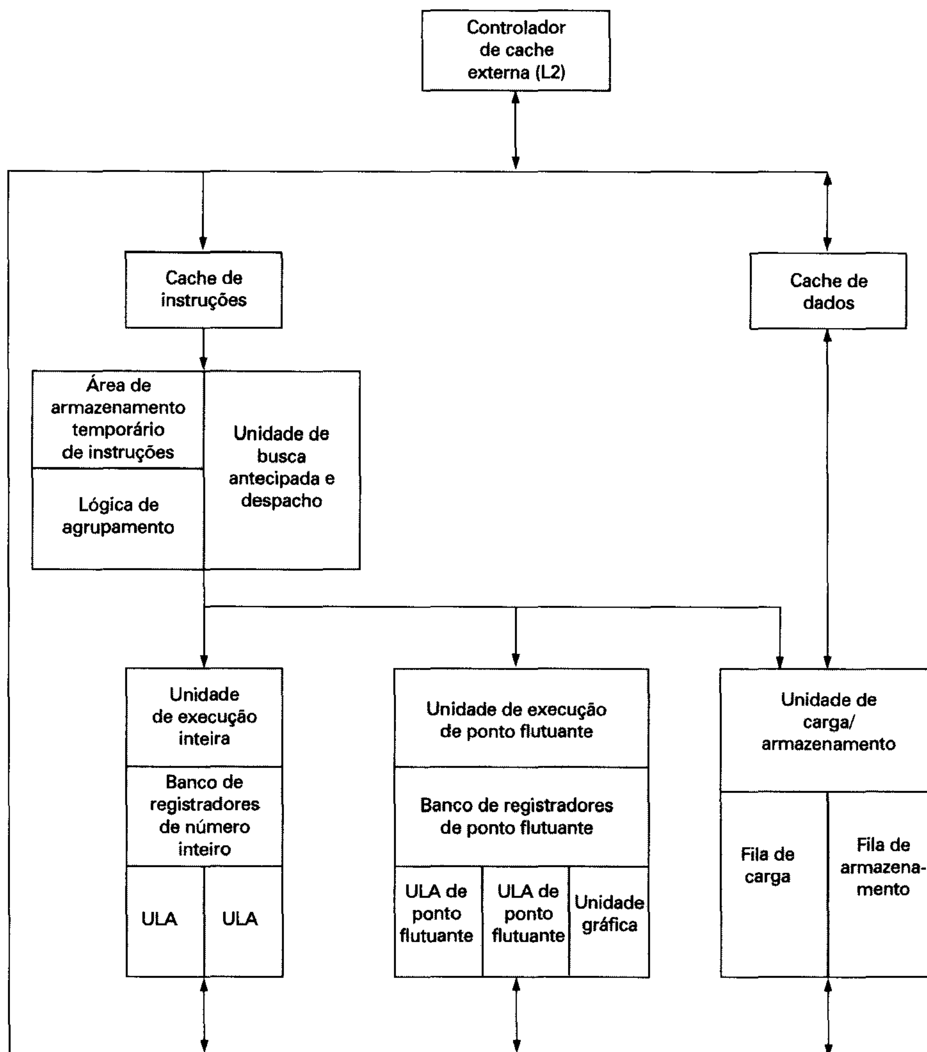


Figura 13.16 Diagrama de bloco do UltraSPARC IIi.

- **N1**: quando ocorre falta na cache, a instrução entra na fila de carga ou na fila de armazenamento para acessar a cache L2 e a memória principal.
- **N2**: esse estágio é simplesmente um estágio de espera para a *pipeline* de ponto flutuante.
- **N3**: as exceções de programa (traps) são tratadas durante esse estágio.
- **Escrita**: todos os resultados de operações são escritos nos bancos de registradores nesse estágio.

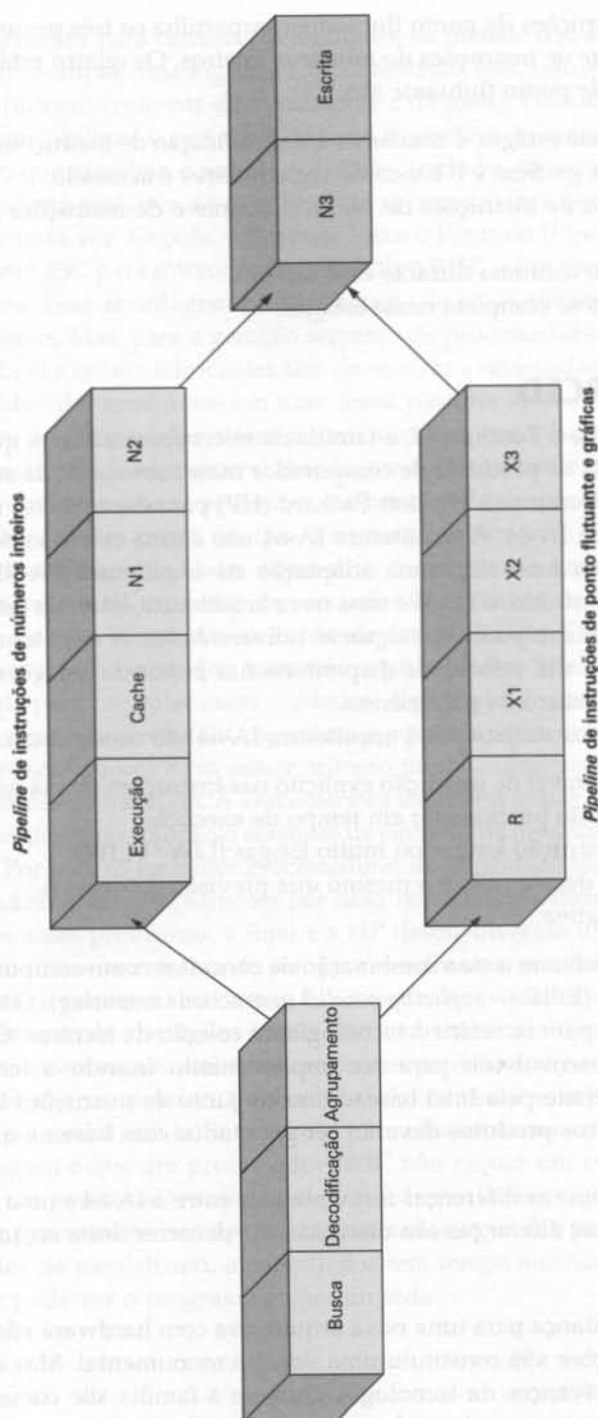


Figura 13.17 *Pipeline* de instruções do UltraSPARC II.

A *pipeline* de instruções de ponto flutuante compartilha os três primeiros e os dois últimos estágios da *pipeline* de instruções de números inteiros. Os quatro estágios exclusivos da *pipeline* de instruções de ponto flutuante são:

- **R**: durante esse estágio, é concluída a decodificação de instruções de ponto flutuante e instruções gráficas e o banco de registradores é acessado.
- **X1**: a execução de instruções de ponto flutuante e de instruções gráficas é iniciada nesse estágio.
- **X2**: a execução continua durante esse estágio.
- **X3**: a execução se completa nesse estágio.

13.7 IA-64/MERCED

Com o Pentium II e o Pentium III, a família de microprocessadores que começou com o 8086 e constituiu a linha de produtos de computador mais bem-sucedida até então parece ter chegado ao fim. A Intel uniu-se à Hewlett-Packard (HP) para desenvolver uma nova arquitetura de 64 bits, chamada IA-64. A arquitetura IA-64 não é uma extensão de 64 bits da arquitetura de 32 bits x86 da Intel nem uma adaptação da arquitetura PA-RISC de 64 bits da Hewlett-Packard. Ao contrário, o IA-64 é uma nova arquitetura, baseada em anos de pesquisa desenvolvida pelas duas companhias e algumas universidades. A arquitetura explora o vasto conjunto de circuitos de alta velocidade disponíveis nas próximas gerações de circuitos integrados com o uso sistemático de paralelismo.

Os conceitos básicos subjacentes à arquitetura IA-64 são os seguintes:

- Paralelismo de nível de instrução explícito nas instruções de máquina, em vez de ser determinado pelo processador em tempo de execução.
- Palavras de instrução longas ou muito longas (LIW / VLIW).
- Predicação de desvio (não é o mesmo que previsão de desvio).
- Carga especulativa.

A Intel e a HP se referem a essa combinação de conceitos como computação de instrução explicitamente paralela (EPIC — *explicitly parallel instruction computing*). O termo **EPIC** é usado pela Intel e pela HP para se referir à tecnologia ou coleção de técnicas. O **IA-64** é um conjunto de instruções desenvolvido para ser implementado usando a tecnologia EPIC. O primeiro produto planejado pela Intel baseado no conjunto de instruções IA-64 tem o nome código de **Merced**. Outros produtos deverão ser projetados com base na mesma arquitetura IA-64.

A Tabela 13.2 resume as diferenças fundamentais entre o IA-64 e uma abordagem superescalar tradicional. Essas diferenças são discutidas no decorrer desta seção.

Motivação

Para a Intel, a mudança para uma nova arquitetura com hardware não compatível com a arquitetura de instruções x86 constituiu uma decisão monumental. Mas ela se viu forçada a dar esse passo pelos avanços da tecnologia. Quando a família x86 começou, no final dos anos 70, uma pastilha de processador tinha dezenas de milhares de transistores e era um dispositivo essencialmente escalar. Isto é, as instruções eram processadas uma de cada vez, não sendo usadas *pipelines* ou com *pipelines* com um pequeno número de estágios. Com o aumento

do número de transistores para centenas de milhares, na metade dos anos 80, a Intel introduziu as *pipelines* (por exemplo, veja Figura 11.18). Enquanto isso, outros fabricantes buscaram tirar vantagem do número crescente de transistores e da maior velocidade por meio da abordagem RISC, que possibilitou o uso mais efetivo de *pipelines* e, mais tarde, da combinação RISC/superescalar, que envolvia o uso de múltiplas unidades de execução. Com o Pentium, a Intel fez uma modesta tentativa de usar técnicas superescalares, permitindo executar duas instruções CISC de cada vez. Depois, o Pentium Pro e o Pentium II incorporaram um mapeamento de instruções CISC para microoperações de tipo RISC e um uso mais agressivo de técnicas superescalares. Essa abordagem possibilitou o uso mais efetivo de uma pastilha com milhões de transistores. Mas, para a geração seguinte de processadores, posterior ao Pentium II e Pentium III, a Intel e outros fabricantes têm de encarar a necessidade de usar efetivamente as dezenas de milhões de transistores em uma única pastilha de processador.

Os projetistas de processadores têm poucas escolhas em relação à forma de usar essa abundância de transistores. Uma possível abordagem é usar os transistores extras para caches internas maiores. Caches maiores tendem a melhorar o desempenho até certo ponto mas, eventualmente, atinge-se um ponto em que o aumento de tamanho da cache resulta em melhoria mínima da taxa de acerto. Outra alternativa é aumentar o grau de processamento superescalar, adicionando mais unidades de execução. O problema dessa abordagem é que os projetistas se vêm, de fato, diante de um limite de complexidade. Quanto maior é o número de unidades de execução adicionadas, tornando o processador "mais largo", mais complexa é a lógica necessária para controlar essas unidades. A previsão de desvio deve ser melhorada, tem de ser usado processamento fora de ordem e *pipelines* maiores devem ser empregadas. Mas, o uso de *pipelines* maiores e em maior número implica uma perda muito maior no caso de uma previsão de desvio incorreta. A execução fora de ordem requer grande número de registradores de renomeação e um complexo conjunto de circuitos de bloqueio, para lidar com dependências de dados. Por isso, os melhores processadores da atualidade são capazes de gerenciar o despacho de, no máximo, quatro instruções por ciclo, normalmente menos.

Para resolver esses problemas, a Intel e a HP desenvolveram uma nova abordagem de projeto, que possibilita o uso efetivo de um processador com muitas unidades de execução paralela. A essência dessa nova abordagem é o conceito de paralelismo explícito. Nessa abordagem, o escalonamento de instruções é feito estaticamente em tempo de compilação, em vez de ser feito dinamicamente pelo processador, em tempo de execução. O compilador determina quais instruções podem ser executadas em paralelo e inclui essa informação na instrução de máquina. O processador usa tal informação para efetuar a execução paralela. Uma vantagem dessa abordagem é que um processador EPIC não requer um conjunto de circuitos tão complexo quanto um processador superescalar, que despacha instruções fora de ordem. Além disso, enquanto o processador tem apenas alguns nanossegundos para determinar as potenciais oportunidades de paralelismo, o compilador tem tempo suficiente para examinar o código com calma e pode ver o programa como um todo.

Tabela 13.2 Arquitetura superescalar tradicional *versus* arquitetura IA-64

Superescalar	IA-64
Instruções de tipo RISC, uma por palavra	Instruções de tipo RISC empacotadas em grupos de três
Múltiplas unidades de execução paralelas	Múltiplas unidades de execução paralelas
Reordena e otimiza o fluxo de instrução em tempo de execução	Reordena e otimiza o fluxo de instruções em tempo de compilação
Previsão de desvio com execução especulativa de um caminho	Execução especulativa nos dois caminhos de um desvio
Carrega dados da memória apenas quando necessários, tentando primeiro encontrar os dados nas caches	Carrega dados especulativamente, antes que sejam necessários, ainda tentando encontrar os dados primeiro nas caches

Organização

Assim como muitas arquiteturas de processador, a arquitetura IA-64 pode ser implementada em uma variedade de organizações. A Figura 13.18 sugere a organização de uma máquina IA-64, em termos gerais. As características fundamentais são:

- **Um grande número de registradores:** o formato de instrução do IA-64 supõe o uso de 256 registradores de 64 bits, 128 para números inteiros, valores lógicos e uso geral e 128 para números de ponto flutuante e uso gráfico. Existem ainda 64 *registradores de predicado* de 1 bit, que são usados para *execução predicada,* como será explicado mais adiante.
- **Múltiplas unidades de execução:** uma máquina comercial superescalar típica provê quatro *pipelines* paralelas, usando quatro unidades de execução paralelas, para números inteiros e números de ponto flutuante. Espera-se que a arquitetura IA-64 seja implementada em sistemas com oito ou mais unidades paralelas.

O banco de registradores é bastante grande, se comparado ao da maioria das máquinas RISC e superescalares. A razão para isso é que é necessário um grande número de registradores para suportar a um alto grau de paralelismo. Em uma máquina superescalar tradicional, a linguagem de máquina (e a linguagem de montagem) emprega um pequeno número de registradores visíveis e o processador mapeia esses registradores em um número maior de registradores físicos, usando técnicas de renomeação de registradores e análise de dependências. Para tornar o paralelismo explícito e aliviar o processador da responsabilidade de renomear registradores e efetuar análise de dependências, precisamos de um grande número de registradores explícitos.

O número de unidades de execução é função do número de transistores disponíveis em uma implementação particular. O processador explora paralelismo até o ponto em que for possível. Por exemplo, se o fluxo de instruções de linguagem de máquina indica que podem ser executadas em paralelo oito instruções de número inteiro, um processador com quatro *pipelines* de número inteiro executará essas instruções em duas etapas. Um processador com oito *pipelines* poderá executar todas as oito instruções simultaneamente.

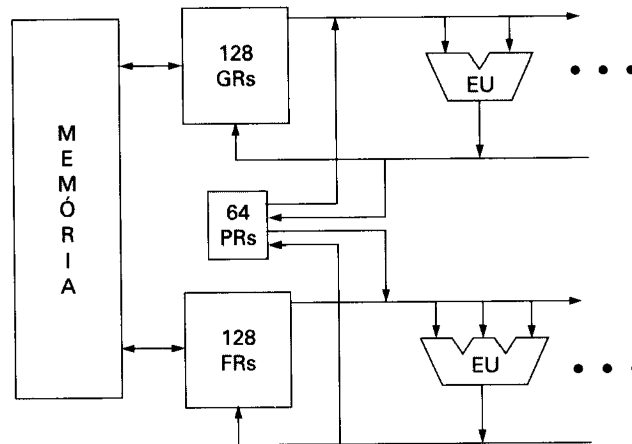


Figura 13.18 Organização geral da arquitetura IA-64.

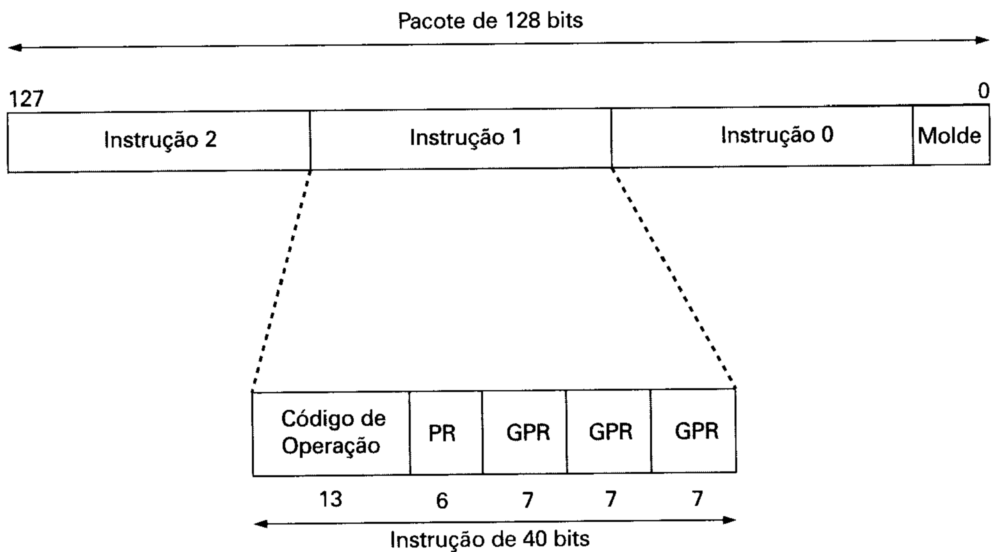


Figura 13.19 Formato de instrução da arquitetura IA-64.

Formato de instrução

O IA-64 define um pacote de instrução de 128 bits, que contém três instruções e um campo de molde (Figura 13.19). O processador pode buscar um pacote de instruções de cada vez; assim, cada busca traz três instruções. O campo de molde contém informação que indica quais instruções podem ser executadas em paralelo. A interpretação do campo de molde não é confinada a um único pacote: o processador pode olhar múltiplos pacotes para determinar instruções que podem ser executadas em paralelo. Por exemplo, o fluxo de instruções pode ser tal que oito instruções possam ser executadas em paralelo. Nesse caso, o compilador reordena as instruções de forma que essas oito instruções sejam colocadas em pacotes contíguos, e atribui valores aos bits de molde de maneira que o processador saiba que as oito instruções são independentes.

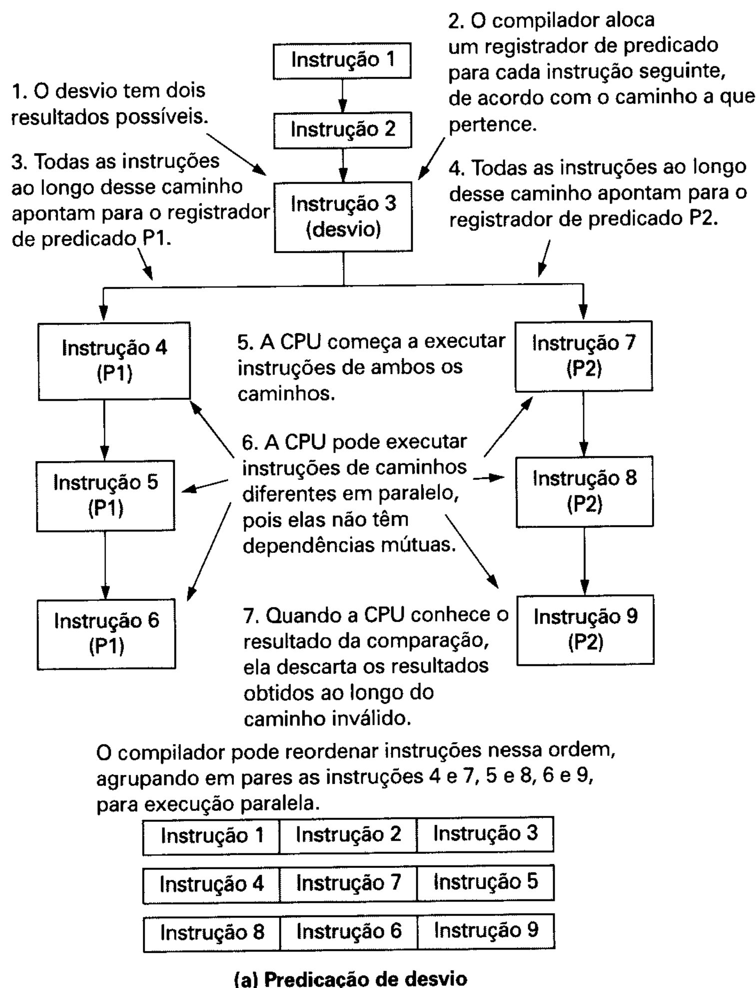
As instruções empacotadas não precisam estar na ordem em que aparecem no programa original. Além disso, devido à flexibilidade dada pelo campo de molde, o compilador pode misturar instruções dependentes e independentes em um mesmo pacote. Diferentemente de alguns projetos VLIW anteriores, no IA-64 não é necessário inserir instruções de operação nula (NOP) para preencher pacotes.

Cada instrução tem formato com tamanho fixo de 40 bits. Esse tamanho é um pouco maior que o tamanho tradicional de 32 bits encontrado em máquinas RISC e em máquinas RISC superescalares (embora seja bem menor que as microoperações de 118 bits do Pentium II). Dois fatores levam aos bits adicionais. Primeiro, a arquitetura IA-64 faz uso de maior número de registradores que uma máquina RISC típica: 128 registradores de número inteiro e 128 registradores de ponto flutuante. Segundo, para acomodar a técnica de execução predicada, uma máquina IA-64 inclui 64 registradores de predicado. Esses registradores são invisíveis para o programador; é o compilador que insere uma referência a um registrador de predicado em uma instrução. Seu uso é explicado na subseção seguinte.

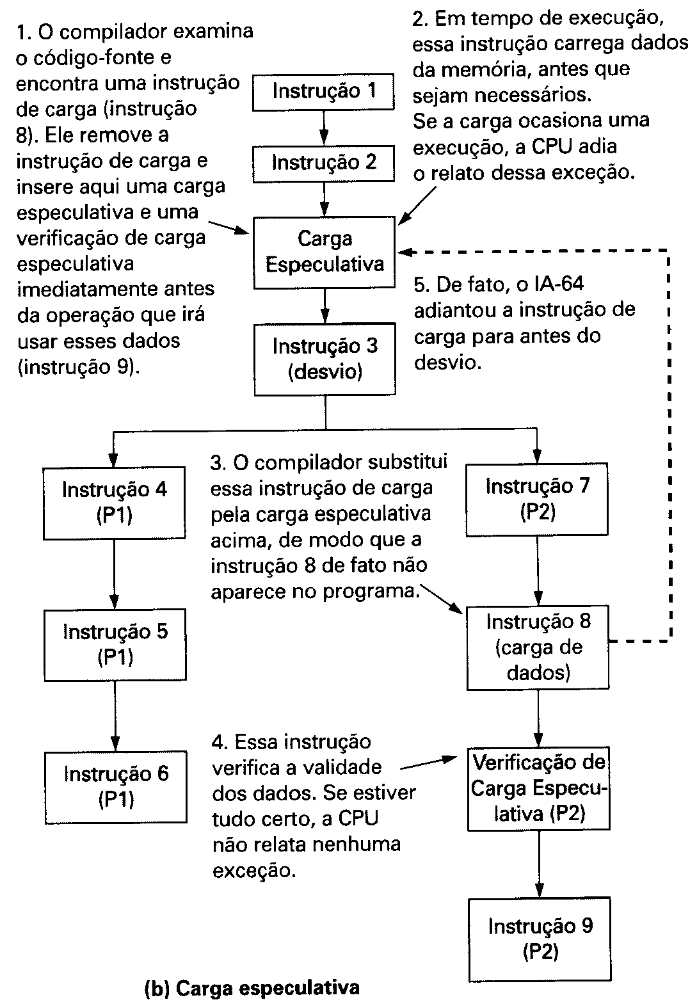
Execução predicada

As duas inovações fundamentais da arquitetura IA-64 são a *execução predicada* e a *carga especulativa* de dados. A Figura 13.20, baseada na figura de Half (1997), ilustra essas duas técnicas, que são discutidas nessa subseção e na próxima.

A predicação é uma técnica em que o compilador determina quais instruções podem ser executadas em paralelo. Nesse processo, o compilador elimina desvios do programa, usando execução condicional. Um exemplo típico em uma linguagem de alto nível é uma instrução **if-then-else**. Um compilador tradicional insere um desvio condicional no ponto correspondente ao **if** dessa construção. Para um dado resultado lógico da condição, o desvio não é tomado e o próximo bloco de instruções, que representa o caminho **then**, é executado; no final desse caminho existe um desvio incondicional que salta o próximo bloco de instruções, que representa o caminho **else**. Se a condição tem o outro resultado lógico, o desvio é tomado em torno do bloco de instruções **then** e a execução continua no bloco de instruções correspondente ao **else**. Os dois fluxos de instrução se juntam à frente, no final do bloco **else**. Um compilador IA-64 faz o seguinte (Figura 13.20a):



(a) Predicação de desvio



(b) Carga especulativa

Figura 13.20 Predicação de desvio e carga especulativa na arquitetura IA-64.

1. No ponto do programa correspondente ao **if**, ele insere uma instrução de comparação que cria dois predicados. Se a comparação for verdadeira, é atribuído valor verdadeiro ao primeiro predicado e falso ao segundo; se a comparação for falsa, é atribuído valor falso ao primeiro e valor verdadeiro ao segundo.
2. A cada instrução do caminho **then** é adicionada uma referência para o registrador de predicado que contém o valor do primeiro predicado; cada instrução do caminho **else** é aumentada com uma referência para o registrador de predicado que contém o valor do segundo predicado.
3. O processador executa instruções ao longo dos dois caminhos. Quando o resultado da comparação é conhecido, o processador descarta os resultados obtidos ao longo de um dos caminhos e confirma os resultados obtidos ao longo do outro caminho.

Como exemplo, considere o seguinte código-fonte:

Código-fonte:

```
if (a&&b)
    j = j + 1;
else
    if (c)
        k = k + 1;
    else
        k = k - 1;
i = i + 1;
```

Os dois comandos if selecionam, juntos, um dentre três caminhos. Esse código pode ser compilado para o seguinte código em linguagem de montagem, que inclui três instruções de desvio condicional e uma instrução de desvio incondicional:

Código em linguagem de montagem

```
     beq a, 0, L1
     beq b, 0, L1
     add j, j, 1
     jump L3
L1:  beq c, 0, L2
     add k, k, 1
     jump L3
L2:  sub k, k, 1
L3:  add i, i, 1
```

A Figura 13.21 mostra o diagrama de fluxo desse código em linguagem de montagem. O diagrama divide o programa em blocos de código separados. A cada bloco que é executado condicionalmente, o compilador pode atribuir um predicado. Esses predicados são indicados na Figura 13.21. Supondo que todos os predicados são inicializados com valor falso, o código resultante é o seguinte:

Código com predicado:

```
(1) P1, P2  = cmp (a == 0)
(2) <P2> P1, P3 = cmp (b == 0)
(3) <P3> add j, j, 1
(4) <P1> P4, P5 = cmp (c != 0)
(5) <P4> add k, k, 1
(6) <P5> sub k, k, 1
(7) add i, i, 1
```

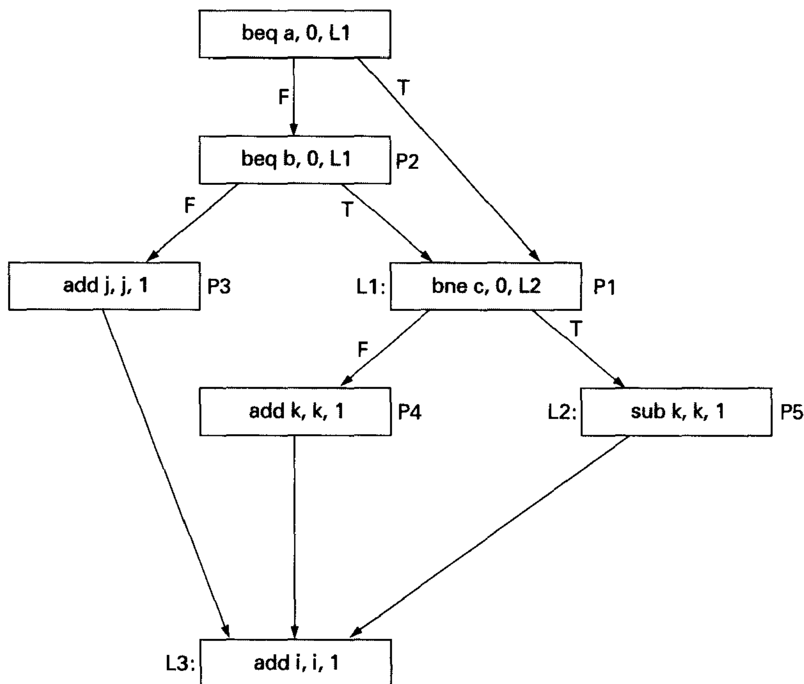


Figura 13.21 Exemplo de predicação de desvio.

Foram introduzidas três construções. Uma instrução da forma:

```
<PI> instrução
```

deve ser executada apenas se o predicado PI for verdadeiro. Em termos do formato de instrução da Figura 13.19, o campo PR tem o valor I, para indicar o registrador de predicado I de 1 bit. O processador busca, decodifica e começa a executar essa instrução, mas apenas decide se deve ou não confirmar os resultados dessa instrução depois de determinar se o valor do registrador de predicado I é 1 (VERDADEIRO) ou 0 (FALSO).

Uma instrução da forma:

```
PJ, PK = cmp (relação)
```

atribui valor VERDADEIRO ao predicado J e FALSO ao predicado K, se a relação for verdadeira, e atribui valor FALSO ao predicado J e VERDADEIRO ao predicado K, se a relação for falsa. Essa instrução não tem formato; é igual ao mostrado na Figura 13.19, que apresenta o formato de instruções predicadas para transferência de dados de registrador para registrador. O formato dessa instrução inclui dois campos de registrador de predicado. O processador busca, decodifica e executa essa instrução, e armazena os resultados nos registradores de predicado J e K.

Finalmente, uma instrução geradora de predicado pode, ela própria, ser também associada a um predicado:

```
<PI> PJ, PK = cmp (relação)
```

Os registradores de predicado J e K são atualizados com base no resultado da comparação, caso o predicado PI seja verdadeiro. Se o predicado PI for falso, a instrução não será executada. O formato dessa instrução requer três campos de registrador de predicado.

Retornando ao nosso programa, os dois primeiros desvios condicionais do código de montagem são traduzidos como duas instruções de comparação predicadas. Se a primeira instrução atribui valor falso a P2, a segunda instrução não é executada. P3 será verdadeiro apenas se a condição do comando if mais externo do código-fonte for verdadeira. Por isso, a parte then do comando if mais externo tem P3 como predicado. A instrução (4) do código predicativo decide se deve ser executada a instrução de adição ou a instrução de subtração que ocorre na parte else do comando if mais externo. Finalmente, o valor de i é incrementado, incondicionalmente. Olhando o código-fonte e o código predicativo, vemos que apenas uma dentre as instruções (3), (5) e (6) deve ser executada. Em um processador superescalar comum, seria usada previsão de desvio para adivinhar qual das três instruções deve ser executada e seguir por esse caminho. Se a previsão fosse errada, as instruções inválidas deveriam ser descarregadas da *pipeline*. Um processador IA-64 pode iniciar a execução dessas três instruções e, uma vez determinados os valores dos registradores de predicado, confirmar apenas os resultados da instrução válida. Portanto, o processador usa unidades de execução paralela adicionais, para evitar atrasos devidos à retirada de instruções inválidas da *pipeline*.

Grande parte das pesquisas originais relativas à execução predicada foi realizada na Universidade de Illinois. Os estudos de simulação feitos pelos pesquisadores indicam que o uso de predicação resulta em substancial redução de desvios dinâmicos e de previsões de desvio incorretas, e em substancial melhoria de desempenho de processadores com múltiplas *pipelines* paralelas (veja, por exemplo, Mahlke e outros, 1994; 1995).

Carga especulativa

Outra inovação fundamental na arquitetura IA-64 é a carga especulativa de dados. Isso habilita o processador a carregar dados da memória antes que sejam necessários no programa, evitando atrasos de latência de memória. Além disso, o processador adia o relato de exceções, até que isso seja necessário. É usado o termo "içar" (*hoist*) para se referir à mudança de posição de uma instrução de carga para um ponto anterior no fluxo de instruções.

Minimizar a latência de carga é crucial para melhorar o desempenho. Tipicamente, no início de um bloco de código, existem diversas operações de carga para trazer dados da memória para registradores. Como a memória, mesmo se acrescida de um ou dois níveis de cache, é mais lenta que o processador superescalar, os atrasos para obter dados na memória tornam-se um gargalo no sistema. A idéia para minimizar esse problema é reordenar as instruções do código de forma que a carga de dados seja efetuada o mais cedo possível. Isso pode ser feito, até certo ponto, pelo compilador. Entretanto, ao se tentar mover uma instrução de carga por meio do fluxo de controle, pode ocorrer um problema. Uma instrução de carga não pode ser movida incondicionalmente para antes de um desvio, pois ela pode, de fato, não ocorrer. Poderíamos então mover a instrução de carga condicionalmente, usando predicados, de modo que os dados sejam buscados na memória mas não sejam armazenados em registradores da arquitetura até que o resultado do predicado seja conhecido. O problema dessa estratégia é que a carga pode falhar: uma exceção de endereço inválido ou de falta de página pode ser gerada. Nesse caso, o processador teria de lidar com a exceção, causando atraso na *pipeline*.

Como podemos então mover uma instrução de carga para antes de um desvio? A solução especificada na arquitetura IA-64 é a carga especulativa, que separa o comportamento da carga (entrega do valor) do comportamento de exceção (Figura 13.20b). Uma instrução de carga do programa original é substituída por duas instruções:

- Uma carga especulativa (ld.s), que executa a busca na memória e efetua a detecção de exceção, mas não ativa exceções (não chama a rotina de sistema operacional que trata a exceção). Essa instrução ld.s é colocada em um ponto adequado do programa, anterior ao desvio.
- Uma instrução de verificação (check.s) é inserida no lugar da instrução de carga original e trata exceções. A instrução check.s pode ser predicada, sendo executada apenas se o predicado for verdadeiro.

Se a instrução ld.s detectar uma exceção, será atribuído valor 1 a um bit especial associado ao registrador-alvo. Se a instrução check.s correspondente for executada, o valor desse bit será verificado e, caso tenha valor 1, a instrução check.s desviará a execução para uma rotina de tratamento de exceção.

Um exemplo usado pela Intel e pela HP para avaliar programas predicados e para ilustrar a carga especulativa é o problema das oito rainhas. O objetivo é arranjar oito rainhas em um tabuleiro de xadrez, de forma que nenhuma rainha ameace qualquer outra. A Figura 13.22 mostra uma possível solução. A linha fundamental do código-fonte, contida em um laço de repetição interno, é a seguinte:

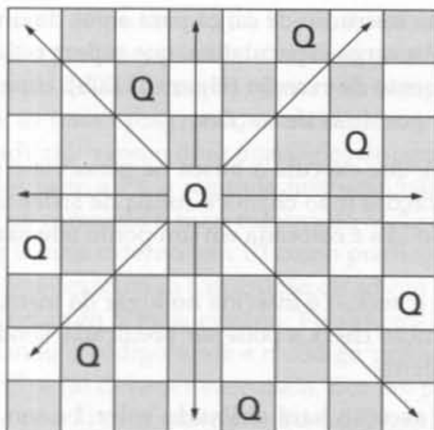
```
if ((b[j] == true) && (a[i+j] == true) && (c[i-j+7] == true))
```

O programa examina todas as colunas, colocando uma rainha em cada uma, de forma que ela não seja atacada por uma rainha colocada anteriormente na linha ou em uma das duas diagonais. O vetor B é usado para verificar as linhas e os vetores A e C, para verificar as duas diagonais.

Um programa correspondente em linguagem de montagem inclui três instruções de carga e três instruções de desvio:

Código em linguagem de montagem

```
(1)       R1 = &b[j]
(2)       ld R2, (R1)
(3)       bne R2, 1, L2
(4)       R3 = &a[i + j]
(5)       ld R4, (R3)
(6)       bne R4, 1, L2
(7)       R5 = &c[i - j + 7]
(8)       ld R6, (R5)
(9)       bne R6, 1, L2
(10) L1: <código para o caminho then>
(11) L2: <código do caminho else>
```




```
Loop:
if ((b[j] == true) && (a[i+j] == true) && (c[i - j+7] == true)
```

Figura 13.22 O problema das oito rainhas.

Usando carga especulativa e execução predicada, temos como resultado o seguinte código:

Código usando carga especulativa e predicação de desvio

```
(1)      R1 = &b[j]
(2)      R3 = &a[i + j]
(3)      R5 = &c[i - j + 7]
(4)      ld R2, (R1)
(5)      ld.s R4, (R3)
(6)      ld.s R6, (R5)
(7)      P1, P2 = cmp (R2 == 1)
(8)      <P2> br L2
(9)      chk.s R4
(10)     P3, P4 = cmp (R4 == 1)
(11)     <P4> br L2
(12)     chk.s R6
(13)     P5, P6 = cmp (R5 == 1)
(14)     <P6> br L2
(15) L1: <código para o caminho then>
(16) L2: <código para o caminho else>
```

O programa em linguagem de montagem é dividido em três blocos básicos de código, cada um dos quais consiste de uma instrução de carga seguida por um desvio condicional. No código em linguagem de montagem, as instruções de cálculo de endereço 4 e 7 constituem cálculos aritméticos simples; como isso pode ser feito em qualquer instante, o compilador move esses cálculos para o início do código. Restam então três blocos simples, cada um dos quais consistindo de uma carga, um cálculo de uma condição e um desvio condicional. Parece haver pouca oportunidade para fazer qualquer execução em paralelo nesse caso. Além disso, se supusermos que uma instrução de carga consome dois ou mais ciclos de relógio, existirá um certo desperdício de tempo antes que o desvio condicional possa ser executado. O que o compilador pode fazer é antecipar a segunda e a terceira instruções de carga (instruções 5 e 8 no código em linguagem de montagem), para antes de todos os desvios. Isso é feito inserin-

do instruções de carga especulativa (instruções 5 e 6) no início do código e instruções de verificação no lugar das instruções de carga originais (instruções 9 e 12).

Essa transformação torna possível executar as três instruções de carga em paralelo, iniciando as cargas de dados mais cedo, para minimizar ou evitar atrasos devidos à latência de operações de carga. O compilador pode ir mais além, fazendo uso mais agressivo de predicação, e eliminar duas das três instruções de desvio:

Código revisado com carga especulativa e predicação de desvio

```
(1)        R1 = &b[j]
(2)        R3 = &a[i + j]
(3)        R5 = &c[i - j + 7]
(4)        ld R2, (R1)
(5)        ld.s R4, (R3)
(6)        ld.s R6, (R5)
(7)        P1, P2 = cmp (R2 == 1)
(8)        <P1> chk.s R4
(9)        <p1> P3, P4 = cmp (R4 == 1)
(10)       <p3> chk.s R6
(11)       <P3> P5, P6 = cmp (R5 ==1)
(12)       <P6> br L2
(13)   L1: <código para o caminho then>
(14)   L2: <código para o caminho else>
```

Já tínhamos uma comparação que gera dois predicados. No código revisado, em lugar de desviar caso o predicado seja falso, o compilador qualifica a execução da verificação e da próxima instrução de comparação para quando o predicado for verdadeiro. A eliminação de dois desvios significa eliminar duas previsões potencialmente incorretas, economia esta que representa mais do que simplesmente eliminar duas instruções.

13.8 LEITURAS RECOMENDADAS E PÁGINAS WEB

O artigo de Johnson (1991) permanece como excelente e relevante referência sobre projetos superescalares. Alguns artigos de revisão sobre esse assunto são de Smith e Sohi (1995) e Sima (1997). Em Jouppi e Wall (1989a), é abordado o paralelismo no nível de instrução, sendo examinadas várias técnicas para maximizar paralelismo e comparando as abordagens superescalar e superpipeline, por meio de simulação.

Em Popescu (1991) é apresentada uma descrição detalhada de uma proposta de máquina superescalar. Esse artigo é também um excelente tutorial sobre questões de projeto relacionadas a políticas de despacho de instruções fora de ordem. Outro sistema superescalar é proposto em Kuga, Murakami e Tomita (1991); esse artigo aborda a maioria das questões importantes no projeto e implementação de uma máquina superescalar. Lee, Kwok e Briggs (1991) examinam técnicas de software que podem ser usadas para melhorar o desempenho de uma máquina superescalar. Um estudo interessante sobre a extensão em que o paralelismo no nível de instrução pode ser explorado em um processador superescalar é apresentado em Wall (1991).

A *pipeline* de instruções do Pentium II é descrita detalhadamente em Shaneey (1998). Uma boa discussão sobre projeto superescalar é encontrada em Papworth (1996), que cobre o Pentium Pro e analisa algumas das questões relacionadas à organização do processador. O Pentium Pro tem a mesma organização superescalar dos processadores Pentium II e Pentium III, mas não inclui o componente MMX.

Um exame detalhado da *pipeline* de instruções do PowerPC 601 é apresentado em Potter e outros (1994). Outra boa descrição pode ser encontrada em Shankey (1995a).

A melhor abordagem sobre o MIPS R10000 é feita em Yeager (1996). Uma boa discussão sobre o UltraSPARC II é feita em Normoyle et al (1998). Uma visão geral da arquitetura IA-64 é apresentada em Dulong (1998) e Hwu (1998) provê uma breve introdução sobre execução predicada.



Sites Web recomendados:

- **Merced/IA-64:** Página da Intel que contém as mais recentes informações sobre a arquitetura IA-64 e o processador Merced.
- **IMPACT:** Página da Universidade de Illinois, onde foi realizada grande parte das pesquisas sobre execução predicada. Estão disponíveis diversos artigos sobre o assunto.

13.9 EXERCÍCIOS

13.1 Quando o término de instruções fora de ordem é usado em um processador superescalar, a retomada de execução após o processamento de uma interrupção fica mais complicada, porque a condição de exceção pode ter sido detectada em uma instrução que produziu seu resultado fora de ordem. Nesse caso, o programa não pode ser reiniciado a partir da instrução seguinte à instrução que gerou a exceção, pois instruções subseqüentes podem já ter sido completadas e, portanto, isso faria com que essas instruções fossem executadas duas vezes. Sugira um mecanismo para lidar com essa situação.

13.2 Considere a seguinte seqüência de instruções, em que a sintaxe consiste de um código de operação seguido de um registrador de destino e de um ou dois registradores fonte:

0	ADD	R3,	R1,	R2
1	LOAD	R6,	[R3]	
2	AND	R7,	R5,	3
3	ADD	R1,	R6,	R0
4	SRL	R7,	R0,	8
5	OR	R2,	R4,	R7
6	SUB	R5,	R3,	R4
7	ADD	R0,	R1,	R10
8	LOAD	R6,	[R5]	
9	SUB	R2,	R1,	R6
10	AND	R3,	R7,	15

Suponha que é usada uma *pipeline* de quatro estágios: busca, decodificação/iniciação, execução e resposta. Suponha que cada um dos estágios da *pipeline* consome um ciclo de relógio, exceto o estágio de execução. O estágio de execução gasta um ciclo para ope-

rações lógicas e para operações aritméticas simples sobre números inteiros, mas consome cinco ciclos para uma instrução de carga (LOAD) de dado da memória.

Se tivermos uma *pipeline* escalar simples, mas que permite execução fora de ordem, podemos construir a seguinte tabela para a execução das sete primeiras instruções:

Instrução	Busca	Decodificação	Execução	Resposta
0	0	1	2	3
1	1	2	4	9
2	2	3	5	6
3	3	4	10	11
4	4	5	6	7
5	5	6	8	10
6	6	7	9	12

As entradas sob cada um dos quatro estágios da *pipeline* indicam o ciclo de relógio no qual cada instrução começa esse estágio. Nesse programa, a segunda instrução ADD (instrução 3) depende da instrução LOAD (instrução 1) para obter um de seus operandos (R6). Como a instrução LOAD consome cinco ciclos de relógio e a lógica de iniciação de instruções encontra a instrução ADD, que depende dela, apenas dois ciclos de relógio depois, a lógica de iniciação de instruções teria de atrasar a instrução ADD por três ciclos de relógio. Com o uso de iniciação de instruções fora de ordem, o processador pode parar a instrução 3 no ciclo de relógio 4, e então iniciar as três instruções independentes seguintes, que entram em execução nos ciclos de relógio 6, 8 e 9. A execução da instrução LOAD é completada no ciclo de relógio 9 e, assim, a instrução dependente ADD pode ser iniciada no ciclo de relógio 10.

a. Complete a tabela precedente.

b. Refaça a tabela supondo que as instruções não podem ser emitidas para execução fora de ordem. Qual é a economia de tempo de processamento obtida com a emissão de instruções fora de ordem?

c. Refaça a tabela supondo uma implementação superescalar que pode manipular duas instruções de cada vez em cada estágio.

13.3 Na fila de instruções da unidade de despacho do PowerPC 601, instruções podem ser despachadas fora de ordem para as unidades de desvio e de aritmética de ponto flutuante, mas instruções de aritmética de número inteiro devem ser despachadas apenas quando chegam ao início da fila. Por que essa restrição?

13.4 Faça uma figura semelhante à Figura 13.14 para os seguintes casos:

a. Previsão de desvio tomado e previsão correta (desvio foi tomado)

b. Previsão de desvio tomado e previsão incorreta (desvio não foi tomado)

13.5 Considere o seguinte programa em linguagem de montagem:

```
I1: Move  R3, R7        /R3 ← (R7)/
I2: Load  R8, (R3)      /R8 ← Memória (R3)/
I3: Add   R3, R3, 4     /R3 ← (R3) + 4/
I4: Load  R9, (R3)      /R9 ← Memória (R3)/
I5: BLE   R8, R9, L3    / Desvia se (R9) > (R8)/
```

Esse programa inclui dependências de tipo escrita-escrita, leitura-escrita e escrita-leitura. Mostre essas dependências.

13.6 A Figura 13.23 mostra um exemplo de organização de um processador superescalar. O processador pode iniciar duas instruções por ciclo, caso não exista conflito de recurso nem dependências de dados. Existem essencialmente duas *pipelines*, com quatro estágios de processamento (busca, decodificação, execução e armazenamento). Cada *pipeline* tem sua própria unidade de busca e decodificação e unidade de armazenamento. Quatro unidades funcionais (multiplicação, adição, unidade lógica e unidade de carga) são disponíveis para uso no estágio de execução e são dinamicamente compartilhadas pelas duas *pipelines*. As duas unidades de armazenamento podem ser usadas dinamicamente pelas duas *pipelines*, dependendo da disponibilidade dessas unidades em cada ciclo particular. Cada *pipeline* usa uma janela de instruções para examinar instruções à frente, com sua própria lógica de busca e decodificação. Essa janela de instruções é usada para implementar a iniciação de instruções fora de ordem.

Considere o seguinte programa a ser executado nesse processador:

I1: Load	R1, A	/ R1 ← Memória (A)/
I2: Add	R2, R1	/ R2 ← (R2) + (R1)/
I3: Add	R3, R4	/ R3 ← (R3) + (R4)/
I4: Mul	R4, R5	/ R4 ← (R4) × (R5)/
I5: Comp	R6	/ R6 ← (R6)/
I6: Mul	R6, R7	/ R3 ← (R3) × (R4)/

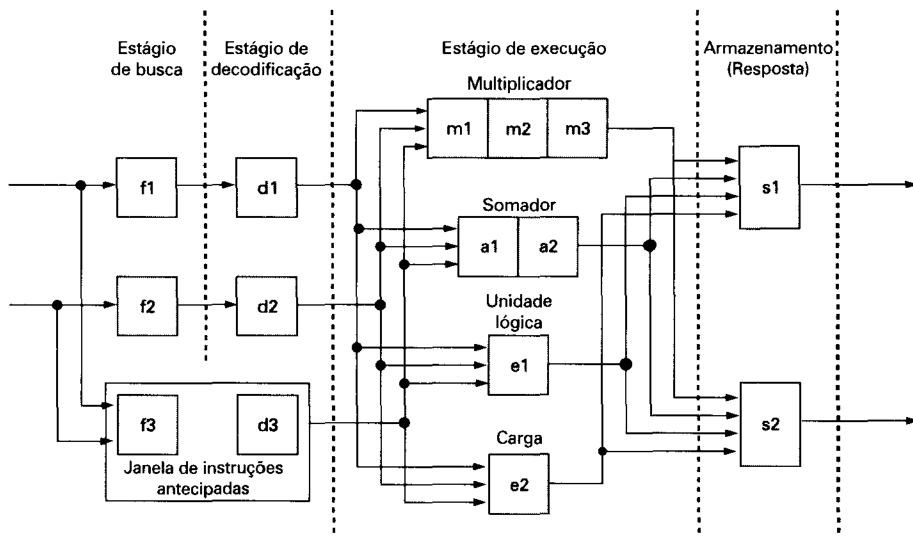


Figura 13.23 Processador superescalar com duas *pipelines*.

a. Que dependências existem no programa?

b. Mostre a atividade da pipeline do processador da Figura 13.23 para esse programa, supondo que é usada política de iniciação e de terminação de execução de instruções em ordem, usando uma representação semelhante à da Figura 13.2.

c. Repita o item (b) para o caso em que é usada política de iniciação em ordem e de terminação de execução fora de ordem.

d. Repita o item (b) para o caso em que é usada política de iniciação fora de ordem e de terminação de execução fora de ordem.

13.7 A Figura 13.24 é extraída de um artigo sobre projetos superescalares. Explique as três partes da figura e defina w, x, y e z.

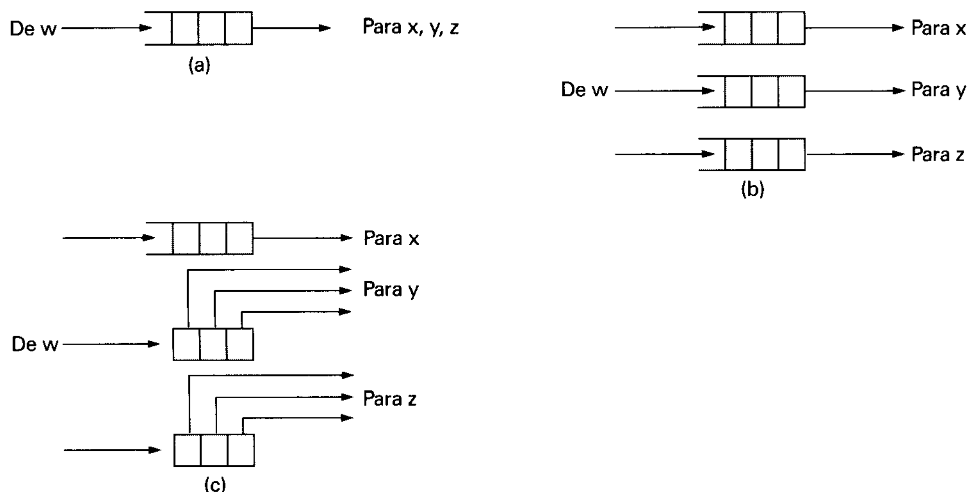


Figura 13.24 Figura para o Exercício 13.7.

13.8 Na Seção 13.7, introduzimos as seguintes construções para execução predicada:

```
PJ, PK = cmp (relação)
<PI> PJ, PK = cmp (relação)
```

Preencha a seguinte tabela-verdade:

PI	Comparação	PJ	PK
não presente	0		
não presente	1		
0	0		
0	1		
1	0		
1	1		

13.9 Para o programa com predicados da Seção 13.7, que implementa o fluxograma da Figura 13.21, indique:

a. As instruções que podem ser executadas em paralelo

b. As instruções que podem ser empacotadas em um mesmo pacote de instruções IA-64

13.10 Considere o seguinte trecho de código-fonte:

```
for  (i = 0; i<100; i++)
     if (A[i]<50)
          j = j + 1;
     else
          k = k + 1;
```

a. Escreva o trecho de código correspondente em linguagem de montagem.

b. Reescreva esse trecho de código em linguagem de montagem usando técnicas de execução predicada.

PARTE 4

A UNIDADE DE CONTROLE

OBJETIVOS DA PARTE 4

Na Parte 3, abordamos as instruções de máquina e as operações efetuadas pelo processador para executar essas instruções. Ficou faltando, nessa discussão, descrever exatamente como é controlada a execução de cada operação individual. Essa tarefa é realizada pela unidade de controle.

A unidade de controle é a parte do processador que controla a execução de instruções. Ela gera sinais de controle externos ao processador para comandar a transferência de dados entre o processador e a memória ou módulos de E/S. Ela gera também sinais de controle internos ao processador para mover dados entre registradores, para comandar a ULA na execução de uma determinada função e para controlar outras operações internas. As entradas para a unidade de controle consistem do registrador de instrução, bits de condição e sinais de controle gerados por fontes externas (por exemplo, sinais de interrupção).

ROTEIRO DA PARTE 4

Capítulo 14 Operação da unidade de controle

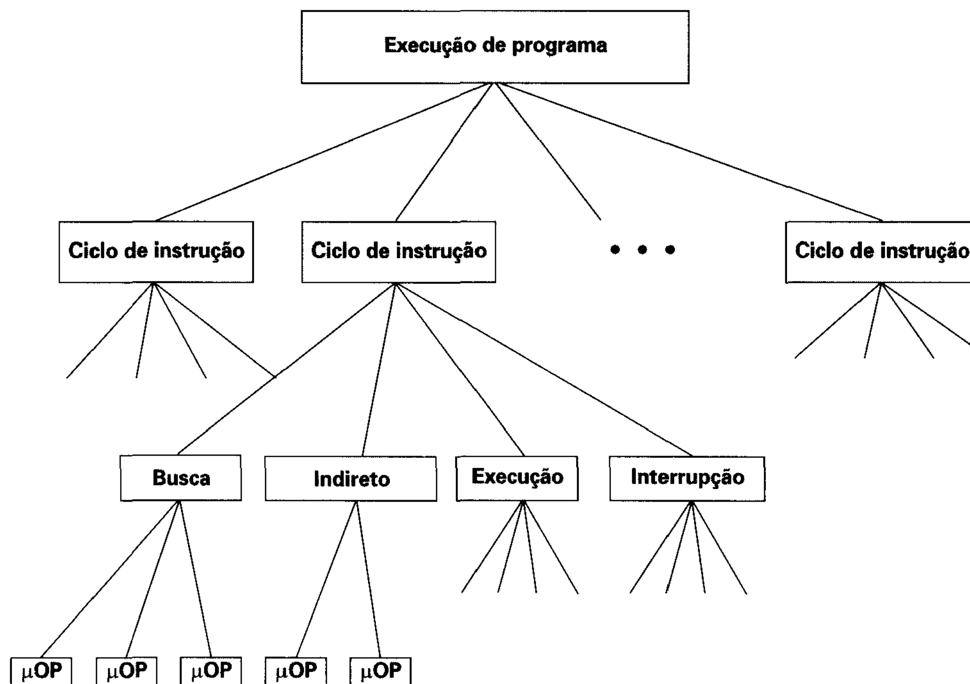
O Capítulo 14 descreve a operação da unidade de controle, explicando o papel dessa unidade em termos funcionais. Veremos que a principal responsabilidade da unidade de controle é determinar a seqüência de execução de operações elementares, denominadas *microoperações*, durante o ciclo de execução de uma instrução.

Capítulo 15 Controle microprogramado

No Capítulo 15, veremos como o conceito de microoperação leva a uma abordagem elegante e poderosa para implementação da unidade de controle, conhecida como microprogramação. Essencialmente, é usada uma linguagem de programação de nível mais baixo: cada instrução de máquina é traduzida como uma seqüência de instruções da unidade de controle de nível mais baixo. Essas instruções de nível mais baixo são denominadas microinstruções e o processo de tradução é chamado microprogramação.

capítulo

14 Operação da Unidade de Controle



- A execução de uma instrução envolve uma seqüência de subpassos, geralmente denominados ciclos. Por exemplo, uma execução pode consistir de ciclos de busca, indireto, execução e interrupção. Cada ciclo é, por sua vez, constituído de uma seqüência de uma ou mais operações fundamentais, denominadas microoperações. Uma única microoperação geralmente envolve uma transferência de dados entre registradores, transferência de dados entre um registrador e um barramento externo ou uma operação simples da ULA.
- A unidade de controle de um processador efetua duas tarefas: (1) faz com que o processador execute microoperações na seqüência apropriada, determinada pelo programa que está sendo executado; e (2) gera sinais de controle que causam a execução de cada microoperação.
- Os sinais de controle gerados pela unidade de controle causam a abertura e o fechamento de portas lógicas, resultando na transferência de dados de e para registradores e na operação da ULA.
- Uma técnica de implementação da unidade de controle é conhecida como implementação por hardware. Nessa técnica, a unidade de controle consiste em um circuito combinatório; seus sinais lógicos de entrada, governados pela instrução de máquina corrente, são convertidos em um conjunto de sinais de controle de saída.

No Capítulo 9, dissemos que existe um longo caminho a ser percorrido desde a especificação do conjunto de instruções de máquina até a definição do processador. Se conhecemos o conjunto de instruções de máquina, compreendendo o efeito de cada código de operação e de cada modo de endereçamento, e conhecemos o conjunto de registradores visíveis para o usuário, já sabemos quais funções o processador deve executar. Mas isso ainda não é tudo. Temos de conhecer também as interfaces externas, usualmente um barramento, e como as interrupções são tratadas. Seguindo essa linha de raciocínio, vemos que, para especificar o funcionamento do processador, precisamos definir o seguinte:

- Operações (códigos de operação)
- Modos de endereçamento
- Registradores
- Interface com módulos de E/S
- Interface com o módulo de memória
- Estrutura de processamento de interrupção

Essa lista, embora geral, é bastante completa. Os itens 1 a 3 são definidos pelo conjunto de instruções. Os itens 4 e 5 tipicamente são definidos pela especificação do barramento do sistema. O item 6 é parcialmente definido pelo barramento do sistema e parcialmente definido pelo tipo de suporte oferecido pelo processador ao sistema operacional.

Pode-se dizer que os itens dessa lista constituem os requisitos funcionais de um processador. Esses itens, que determinam *o que* o processador deve fazer, foram tratados nas Partes 2 e 3. Aqui, abordamos *como* essas funções são implementadas. Mais especificamente, como os vários elementos do processador são controlados para prover essas funções. Em outras palavras, passamos a discutir a unidade de controle, que controla a operação do processador.

14.1 MICROOPERAÇÕES

Vimos que a operação de um computador ao executar um programa consiste de uma seqüência de ciclos de instrução, em que é executada uma instrução de máquina a cada ciclo. Devemos nos lembrar que essa seqüência de ciclos de instrução não corresponde necessariamente à seqüência em que as instruções aparecem no código do programa, devido à existência de instruções de desvio. Portanto, referimo-nos aqui à seqüência de execução de instruções ao longo do tempo.

Vimos, também, que cada ciclo de instrução pode ser considerado como sendo composto de um certo número de passos menores. Uma subdivisão conveniente consiste de ciclos de busca, indireto, execução e interrupção, em que apenas os ciclos de busca e de execução estão sempre presentes.

Entretanto, para projetar uma unidade de controle, precisamos refinar ainda mais essa subdivisão. Na discussão sobre *pipeline* de instruções no Capítulo 11, vimos que é possível decompor ainda mais o ciclo de instrução. De fato, veremos que cada um dos ciclos menores envolve uma série de passos, cada um dos quais envolvendo registradores do processador. Esses passos são chamados de *microoperações.* O prefixo *micro* refere-se ao fato de que cada passo é muito simples. A Figura 14.1 mostra a relação entre os vários conceitos discutidos. Para resumir, a execução de um programa consiste na execução de uma seqüência de instruções. Cada instrução é executada durante um ciclo de instrução, que é constituído de subciclos

menores (por exemplo, busca, indireto, execução, interrupção). A execução de cada subciclo envolve uma ou mais operações simples, isto é, as microoperações.

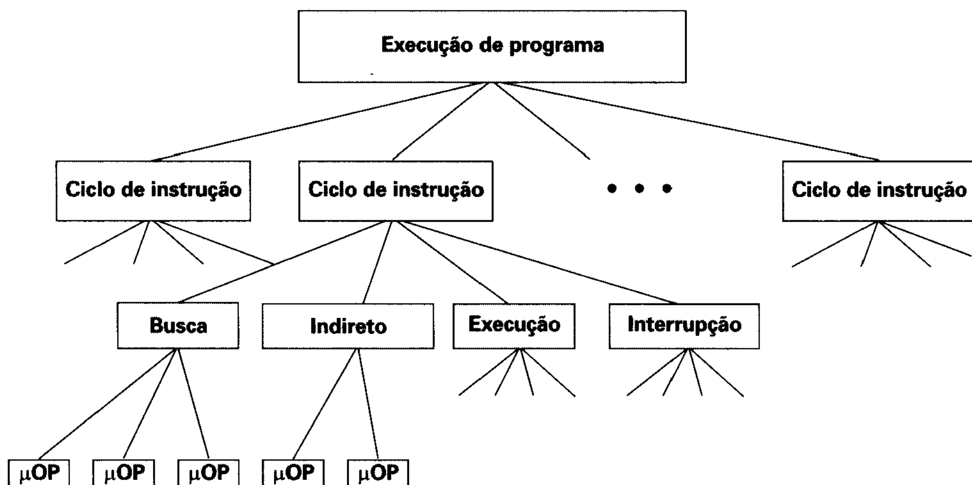


Figura 14.1 Elementos constituintes da execução de um programa.

O ciclo de busca

Começaremos examinando o ciclo de busca, que ocorre no início de cada ciclo de instrução, fazendo com que a instrução seja obtida da memória. Nessa discussão, supomos a organização mostrada na Figura 11.7. Quatro registradores estão envolvidos:

- **Registrador de endereço de memória (MAR):** conectado às linhas de endereço do barramento de sistema. Especifica o endereço de memória para uma operação de leitura ou de escrita.
- **Registrador de armazenamento temporário de dados (MBR):** conectado às linhas de dados do barramento de sistema. Contém um valor a ser armazenado na memória ou o último valor lido da memória.
- **Contador de programa (PC):** mantém o endereço da próxima instrução a ser buscada na memória.
- **Registrador de instrução (IR):** mantém a última instrução buscada na memória.

Observemos a seqüência de eventos de um ciclo de busca, do ponto de vista do seu efeito sobre os registradores do processador. Um exemplo é mostrado na Figura 14.2. No início do ciclo de busca, o endereço da próxima instrução a ser executada está no contador de programa (PC); nesse caso, o endereço é 1100100. O primeiro passo é mover esse endereço para o registrador de endereço de memória (MAR), pois ele é o único registrador conectado às linhas de endereço do barramento do sistema. O segundo passo é trazer a instrução. O endereço desejado (contido no MAR) é colocado no barramento de endereço; a unidade de controle gera um comando de leitura no barramento de controle e o resultado aparece no barramento de dados e é copiado no registrador de armazenamento temporário de dados (MBR). Também é necessário incrementar o contador de instruções (PC), deixando-o pronto para a próxima instrução. Como essas duas ações (ler da memória e somar I ao PC) não interferem uma na outra, elas podem ser realizadas simultaneamente, economizando tempo de processamento.

O terceiro passo é mover o conteúdo do MBR para o registrador de instrução (IR). Isso libera o MBR para ser usado durante um possível ciclo de endereçamento indireto à memória.

MAR	
MBR	
PC	0000000001100100
IR	
AC	

(a) Início

MAR	0000000001100100
MBR	
PC	0000000001100100
IR	
AC	

(b) Primeiro Passo

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100110
IR	
AC	

(c) Segundo Passo

MAR	0000000001100100
MBR	0001000000100000
PC	0000000001100110
IR	0001000000100000
AC	

(d) Terceiro Passo

Figura 14.2 Seqüência de eventos do ciclo de busca.

Portanto, um simples ciclo de busca consiste, de fato, de três passos e quatro microoperações. Cada microoperação envolve movimentação de dados de ou para um registrador. Se essas movimentações de dados não interferem umas com as outras, várias delas podem ser efetuadas em um único passo, para economizar tempo de processamento. Simbolicamente, essa seqüência de eventos pode ser escrita como a seguir, onde *I* é o tamanho de uma instrução:

$$
\begin{aligned}
t_1&: \text{MAR} \leftarrow (\text{PC}) \\
t_2&: \text{MBR} \leftarrow \text{Memória} \\
&\quad \text{PC} \leftarrow (\text{PC}) + I \\
t_3&: \text{IR} \leftarrow (\text{MBR})
\end{aligned}
$$

Diversos comentários precisam ser feitos sobre essa seqüência. Supõe-se que existe um relógio, para propósitos de temporização, e que ele emite pulsos de relógio regularmente espaçados. Cada pulso de relógio define uma unidade de tempo. Portanto, todas as unidades de tempo têm a mesma duração. Cada microoperação é executada dentro do intervalo de uma única unidade de tempo. A notação (t_1, t_2, t_3) representa unidades de tempo sucessivas. Em outras palavras, temos:

- **Primeira unidade de tempo**: move o conteúdo de PC para MAR.
- **Segunda unidade de tempo**: move o conteúdo da posição de memória especificada por MAR para o MBR. Incrementa de *I*: o valor de PC.
- **Terceira unidade de tempo**: move o conteúdo de MBR para IR.

Note que a segunda e a terceira microoperações são ambas efetuadas durante a segunda unidade de tempo. A terceira microoperação também poderia ter sido agrupada com a quarta, sem que a operação de busca fosse afetada:

t_1: MAR ← (PC)
t_2: MBR ← Memória
t_3: PC ← (PC) + *I*
IR ← (MBR)

Os agrupamentos de microoperações seguem duas regras simples:

1. A seqüência apropriada de eventos deve ser seguida. Portanto, (MAR ← (PC)) deve preceder (MBR ← Memória), uma vez que uma operação de leitura na memória faz uso do endereço contido no MAR.
2. Devem ser evitados conflitos. Não se deve tentar ler e escrever em um mesmo registrador durante a mesma unidade de tempo, pois o resultado é imprevisível. Por exemplo, as microoperações (MBR ← Memória) e (IR ← MBR) não devem ocorrer durante a mesma unidade de tempo.

Um último ponto a ser notado é que uma das microoperações envolve uma adição. Para evitar duplicação de circuito, essa adição poderia ser feita pela ULA. O uso da ULA pode envolver microoperações adicionais, dependendo da funcionalidade da ULA e da organização do processador. Essa discussão será adiada para um ponto mais à frente, neste capítulo.

É útil comparar os eventos descritos acima e nas próximas subseções com a Figura 3.5. Embora as microoperações tenham sido ignoradas nessa figura, a discussão anterior mostra que as microoperações são necessárias para efetuar os subciclos do ciclo de instrução.

O ciclo indireto

Uma vez que a instrução tenha sido buscada, o próximo passo é buscar os operandos fonte. Continuando com o nosso exemplo simples, suponha um formato de instrução de um endereço, que permita endereçamento direto ou indireto. Se a instrução especifica um endereço indireto, um ciclo indireto deve preceder o ciclo de execução. O fluxo de dados difere um pouco do indicado na Figura 11.8 e inclui as seguintes microoperações:

t_1: MAR ← (IR(Endereço))
t_2: MBR ← Memória
t_3: IR(Endereço) ← (MBR(Endereço))

O campo de endereço da instrução é transferido para o MAR e então usado para obter o endereço do operando. Finalmente, o campo de endereço contido em IR é atualizado com o valor do MBR, de modo que ele agora contém um endereço direto, e não um endereço indireto.

O estado do registrador IR está agora igual ao que ele teria se não fosse usado endereçamento indireto, e ele está pronto para o ciclo de execução. Por enquanto, vamos saltar este ciclo e considerar, primeiro, o ciclo de interrupção.

O ciclo de interrupção

Depois de completado o ciclo de execução, é feito um teste para determinar se ocorreu alguma interrupção habilitada. Em caso afirmativo, ocorre um ciclo de interrupção. A natureza desse ciclo varia muito de uma máquina para outra. Apresentamos a seguir uma seqüência de eventos muito simples, como ilustrado na Figura 11.9:

t_1: MBR ← (PC)
t_2: MAR ← Endereço de Salvamento
PC ← Endereço de Rotina
t_3: Memória ← (MBR)

No primeiro passo, o conteúdo de PC é transferido para o MBR, para que possa ser salvo para o retorno da interrupção. Em seguida, o MAR é carregado com o endereço onde o conteúdo do PC deve ser salvo e o PC é carregado com o endereço de início da rotina de tratamento de interrupção. Essas duas ações podem constituir, cada uma, uma microoperação. Entretanto, como a maioria dos processadores inclui múltiplos tipos e/ou níveis de interrupção, podem ser necessárias uma ou várias microoperações adicionais para obter o endereço onde o PC deve ser salvo, e o endereço da rotina de tratamento de interrupção, antes que se possa transferi-los para o MAR e o PC, respectivamente. De qualquer modo, uma vez que isso tenha sido feito, o passo final é armazenar na memória o MBR, que contém o antigo valor do PC. O processador estará então pronto para iniciar o próximo ciclo de instrução.

O ciclo de execução

Os ciclos de busca, indireto e de interrupção são simples e previsíveis. Cada qual envolve uma seqüência de microoperações pequena e fixa e, em cada caso, as mesmas microoperações são repetidas toda vez que o ciclo é executado.

Isso não ocorre no caso do ciclo de execução. Em uma máquina com *N* códigos de operação distintos, poderão existir *N* diferentes seqüências de microoperações. Vamos considerar alguns exemplos hipotéticos.

Primeiramente, considere uma instrução de adição:

```
ADD R1, X
```

que soma o conteúdo de uma posição de memória X ao registrador R1. A seguinte seqüência de microoperações pode ocorrer:

t_1: MAR ← (IR(Endereço))
t_2: MBR ← Memória
t_3: R1 ← (R1) + (MBR)

Começamos com IR que contém a instrução ADD. No primeiro passo, o campo de endereço contido em IR é carregado no registrador MAR. Em seguida, a posição de memória referenciada é lida. Finalmente, os conteúdos de R1 e do MBR são somados pela ULA. Esse é um exemplo simplificado. Podem ser requeridas microoperações adicionais para extrair de IR a especificação do registrador referenciado e, talvez, para armazenar as entradas e a saída da ULA em registradores intermediários.

Vamos agora examinar dois exemplos um pouco mais complexos. Uma instrução bastante comum consiste em incrementar um valor e saltar a próxima instrução na seqüência, caso o valor obtido seja igual a zero:

```
ISZ X
```

O conteúdo da posição de memória X é incrementado de 1. Se o resultado for igual a 0, a próxima instrução será saltada. Uma possível seqüência de microoperações para implementar essa instrução seria:

t_1: MAR ← (IR(Endereço))
t_2: MBR ← Memória
t_3: MBR ← (MBR) + 1
t_4: Memória ← (MBR)
If ((MBR) = 0) then (PC ← (PC) + I)

A característica nova introduzida aqui é a ação condicional: o registrador PC é incrementado se (MBR) = 0. O teste e a ação correspondente podem ser implementados como uma única microoperação. Note também que essa microoperação pode ser efetuada durante a mesma unidade de tempo em que o valor atualizado de MBR é armazenado de volta na memória.

Finalmente, vamos considerar uma instrução de chamada de sub-rotina. Como exemplo, considere a seguinte instrução para desviar e salvar um endereço:

```
BSA X
```

O endereço da instrução que segue a instrução BSA é salvo na posição de memória *X*, e a execução do programa reinicia a partir da instrução de endereço *X* + I. O endereço salvo é usado, mais tarde, para o retorno da sub-rotina. Essa é uma técnica direta para prover chamadas de sub-rotinas. A seguinte seqüência de microoperações é suficiente para implementar essa instrução:

t_1: MAR ← (IR(Endereço))
MBR ← (PC)
t_2: PC ← (IR(Endereço))
Memória ← (MBR)
t_3: PC ← (PC) + I

O endereço contido em PC no início da instrução é o endereço da próxima instrução na seqüência. Esse endereço é salvo na posição de memória cujo endereço é designado por IR. Esse endereço é também incrementado, para prover o endereço da instrução a ser usada no próximo ciclo de instrução.

O ciclo de instrução

Vimos que cada fase do ciclo de instrução pode ser decomposta em uma seqüência de microoperações elementares. No nosso exemplo, existe uma única seqüência de operações para os ciclos de busca, indireto e de interrupção, e, para o ciclo de execução, existe uma seqüência de microoperações para cada código de operação.

Para completar a cena, precisamos juntar essas seqüências de microoperações, tal como é feito na Figura 14.3. Suponhamos que exista um registrador adicional de 2 bits, denominado *código de ciclo de instrução* (ICC). O registrador ICC designa o estado do processador em termos do subciclo do ciclo de instrução que ele está executando:

00: Busca
01: Indireto
10: Execução
11: Interrupção

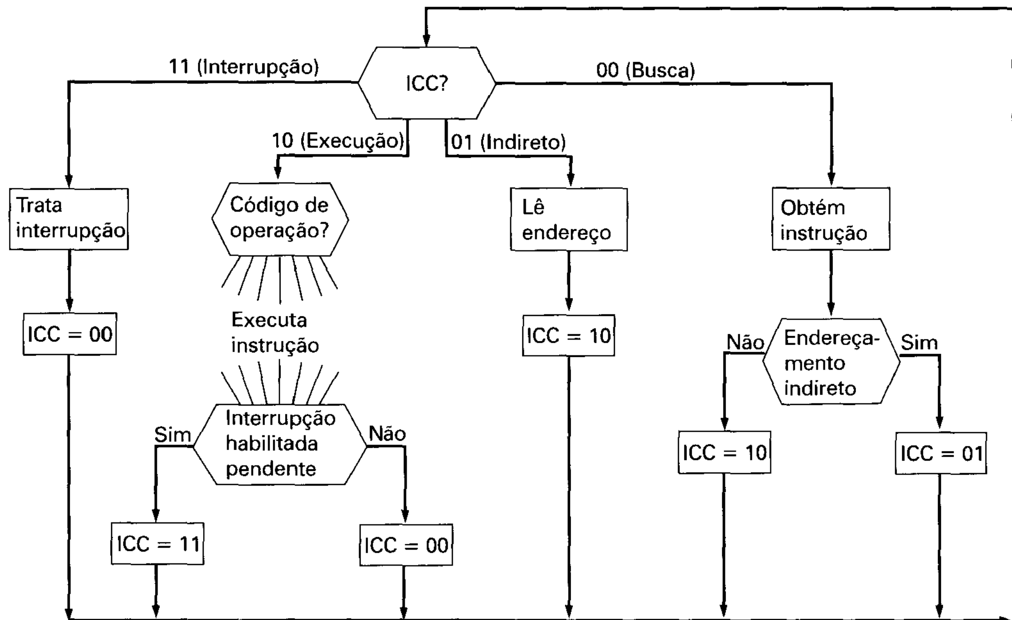


Figura 14.3 Fluxograma do ciclo de instrução.

Ao final de cada um dos quatro ciclos, o registrador ICC é atualizado apropriadamente. O ciclo indireto é sempre seguido por um ciclo de execução. O ciclo de interrupção é sempre seguido por um ciclo de busca (veja Figura 11.5). Tanto no ciclo de execução como no ciclo de busca, o próximo ciclo depende do estado do sistema.

O fluxograma apresentado na Figura 14.3 define portanto a seqüência completa de microoperações, dependendo apenas da seqüência de instruções e do padrão de ocorrência de interrupções. É claro que esse é um exemplo simplificado. O fluxograma para um processador real seria bem mais complexo. Nesse ponto da nossa discussão, já sabemos que a operação do processador é definida pela execução de uma seqüência de microoperações. Podemos agora examinar como a unidade de controle faz com que essa seqüência de operações ocorra.

14.2 CONTROLE DO PROCESSADOR

Requisitos funcionais

Como resultado da análise apresentada na seção precedente, o comportamento ou o funcionamento do processador foi decomposto em operações elementares, denominadas microoperações. Reduzindo a operação do processador ao seu nível mais fundamental, podemos definir exatamente o que a unidade de controle deve fazer para causar a execução dessas operações elementares. Podemos portanto definir os *requisitos funcionais* da unidade de controle, ou seja, as funções que a unidade de controle deve ser capaz de efetuar. A definição desses requisitos funcionais constitui a base para o projeto e a implementação da unidade de controle.

Com essa informação em mãos, o seguinte processo de três passos leva a uma caracterização da unidade de controle:

1. Defina os elementos básicos do processador.
2. Descreva as microoperações que o processador deve executar.
3. Determine as funções que a unidade de controle deve realizar para causar a execução das microoperações.

Já completamos os dois primeiros passos acima. Nossos resultados podem ser resumidos como a seguir. Primeiro, os elementos básicos do processador são:

- ULA
- Registradores
- Caminhos de dados internos
- Caminhos de dados externos
- Unidade de controle

Um pouco de raciocínio deverá convencê-lo de que essa lista é completa. A ULA constitui a essência funcional do computador. Os registradores são usados para armazenar dados de uso interno no processador. Alguns registradores contêm informação de estado necessária para gerenciar o seqüenciamento de instruções (por exemplo, a palavra de estado do programa). Outros contêm dados que são enviados ou recebidos da ULA, da memória ou de módulos de E/S. Caminhos de dados internos são usados para mover dados entre os registradores e entre um registrador e a ULA. Caminhos de dados externos ligam os registradores à memória e aos módulos de E/S, freqüentemente por meio de um barramento de sistema. A unidade de controle causa a execução de operações dentro do processador.

A execução de um programa consiste de operações envolvendo esses elementos do processador. Como vimos, essas operações consistem de seqüências de microoperações. Revendo a Seção 14.1, você poderá ver que todas as microoperações caem em uma das seguintes categorias:

- Transferência de dados de um registrador para outro.
- Transferência de dados de um registrador para uma interface externa (por exemplo, o barramento de sistema).
- Transferência de dados de uma interface externa para um registrador.
- Execução de operação lógica ou aritmética, usando registradores como entrada e saída.

Todas as microoperações necessárias para efetuar um ciclo de instrução, incluindo todas as microoperações requeridas para executar qualquer instrução do conjunto de instruções da máquina, caem sobre uma dessas categorias.

Podemos agora ser um pouco mais explícitos em relação ao modo como funciona a unidade de controle. A unidade de controle desempenha duas tarefas básicas:

- **Seqüenciamento**: a unidade de controle dirige o processador na execução de uma série de microoperações, na seqüência apropriada, com base no programa que está sendo executado.
- **Execução**: a unidade de controle faz com que cada microoperação seja executada.

Essa descrição funcional define *o que* faz a unidade de controle. O princípio-chave de *como* a unidade de controle opera é o uso de sinais de controle.

Sinais de controle

Definimos os elementos que constituem o processador (ULA, registradores, caminhos de dados) e as microoperações que são executadas. Para que a unidade de controle desempenhe sua função, ela deve ter entradas que lhe possibilitem determinar o estado do sistema e saídas que lhe possibilitem controlar o comportamento do sistema. Essas são as especificações externas da unidade de controle. Internamente, a unidade de controle deve conter a lógica necessária para efetuar suas funções de seqüenciamento e execução de microoperações. Uma discussão sobre a operação interna da unidade de controle será apresentada na Seção 14.3 e no Capítulo 15. O restante desta seção trata da interação entre a unidade de controle e os demais elementos do processador.

A Figura 14.4 constitui um modelo genérico de unidade de controle, mostrando suas entradas e saídas. As entradas são:

- **Relógio**: é a forma pela qual a unidade de controle "marca o tempo". A unidade de controle faz com que uma microoperação (ou um conjunto de microoperações simultâneas) seja executada a cada pulso do relógio, também chamado de ciclo do processador ou de ciclo de relógio.
- **Registrador de instrução**: o código de operação da instrução corrente é usado para determinar as microoperações que devem ser executadas durante o ciclo de execução.
- **Códigos de condição**: essa informação é requerida pela unidade de controle para determinar o estado do processador e a saída de operações previamente executadas pela ULA. Por exemplo, na instrução incrementa-e-salta-se-zero (ISZ), a unidade de controle incrementa o registrador PC, se o código de condição indicador de zero tiver valor 1.
- **Sinais de controle do barramento de controle**: a parte de controle do barramento de sistema fornece sinais para a unidade de controle, tais como sinais de interrupção e de reconhecimento.

As saídas da unidade de controle são:

- **Sinais de controle internos ao processador**: esses sinais são de dois tipos: os que causam movimentação de dados de um registrador para outro e os que ativam funções específicas da ULA.
- **Sinais de controle para o barramento de controle**: existem também dois tipos: sinais de controle para a memória e sinais de controle para os módulos de E/S.

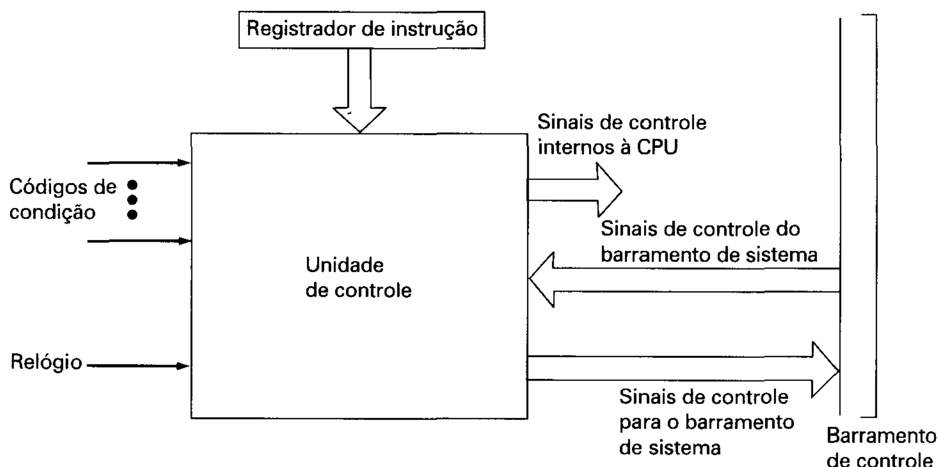


Figura 14.4 Modelo da unidade de controle.

O elemento novo que foi introduzido nessa figura é o sinal de controle. Três tipos de sinais de controle são usados: os que ativam uma função da ULA, os que ativam um caminho de dados e os sinais para o barramento externo do sistema ou para alguma outra interface externa. Todos esses sinais são aplicados, em última instância, como entradas binárias para portas lógicas individuais.

Considere novamente o ciclo de busca, para ver como é feito o controle pela unidade de controle. A unidade de controle mantém informação sobre o passo do ciclo de instrução que está sendo executado. Em um dado ponto, ela saberá que o próximo ciclo a ser executado é um ciclo de busca. O primeiro passo é transferir o conteúdo do registrador PC para o registrador MAR. A unidade de controle faz isso ativando o sinal de controle que abre as portas lógicas entre os bits do PC e os bits do MAR. O próximo passo é ler a palavra da memória para o MBR e incrementar o PC. Isso é feito enviando simultaneamente os seguintes sinais de controle:

- Um sinal de controle que abre portas lógicas, permitindo a transferência do conteúdo do registrador MAR para o barramento de endereço.
- Um sinal de leitura na memória, colocado no barramento de controle.
- Um sinal de controle que abre portas lógicas, permitindo que o conteúdo do barramento de dados seja transferido para o registrador MBR.
- Um sinal de controle para comandar a operação de somar 1 ao conteúdo de PC e armazenar o resultado no próprio PC.

Em seguida, a unidade de controle envia sinais para abrir portas lógicas entre o registrador MBR e o registrador IR.

Isso completa o ciclo de busca, exceto por uma coisa: a unidade de controle deve decidir se realiza, em seguida, um ciclo de endereçamento indireto ou um ciclo de execução. Para decidir isso, ela examina o conteúdo de IR, para determinar se é feita uma referência indireta à memória.

Os ciclos de endereçamento indireto e de interrupção funcionam de maneira similar. No ciclo de execução, a unidade de controle começa examinando o código de operação e, com base nesse código, decide qual seqüência de microoperações deve ser realizada nesse ciclo.

Um exemplo de sinais de controle

Para ilustrar o funcionamento da unidade de controle, vamos examinar um exemplo simples, conforme a Figura 14.5. Ela mostra um processador simples, com um único registrador acumulador. Os caminhos de dados entre seus elementos são indicados na figura. Os caminhos de sinais de controle que emanam da unidade de controle não são mostrados, mas as terminações dos sinais de controle são rotuladas por C_i e indicadas por um pequeno círculo. A unidade de controle recebe sinais de entrada do relógio, do registrador de instrução e dos códigos de condição. Em cada ciclo de relógio, a unidade de controle lê todas as entradas e gera um conjunto de sinais de controle. Esses sinais são dirigidos para três destinos separados:

- **Caminhos de dados**: a unidade de controle controla o fluxo interno de dados. Por exemplo, ao obter uma instrução, o conteúdo do registrador de armazenamento temporário de dados é transferido para o registrador de instrução. Existe uma porta para cada caminho a ser controlado (indicada por um pequeno círculo na figura). Um sinal de controle abre a porta temporariamente, permitindo a passagem do dado.
- **ULA**: A unidade de controle controla a operação da ULA por meio de um conjunto de sinais de controle. Esses sinais ativam vários dispositivos lógicos e portas dentro da ULA.
- **Barramento de sistema**: a unidade de controle envia sinais para as linhas de controle do barramento de sistema (por exemplo, um sinal de leitura na memória).

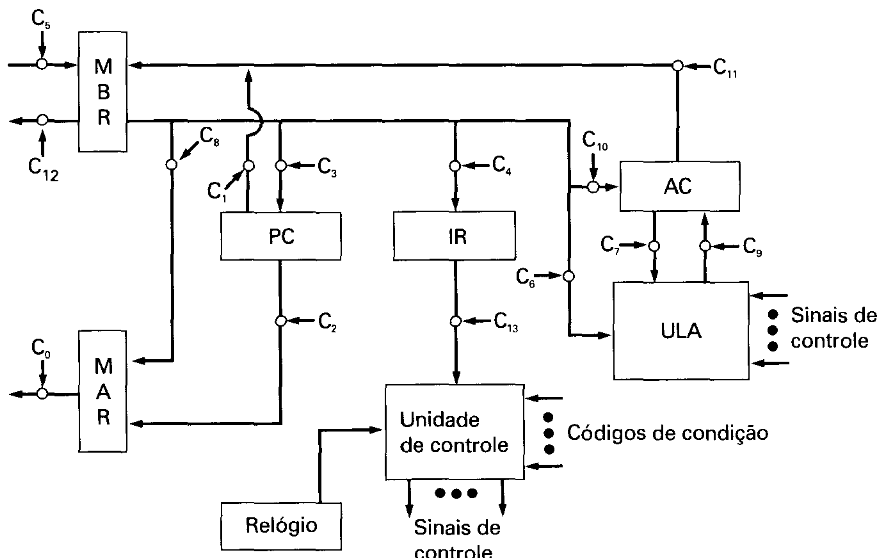


Figura 14.5 Caminhos de dados e sinais de controle.

A unidade de controle mantém informação sobre o passo do ciclo de instrução que está sendo executado. Usando essa informação e lendo seus sinais de entrada, ela gera uma seqüência de sinais de controle, que causam a execução de microoperações. A unidade de controle usa os pulsos do relógio para organizar a execução da seqüência de eventos, garantindo que o tempo entre dois eventos seja suficiente para que os sinais se estabilizem. A Tabela 14.1 indica os sinais de controle requeridos para algumas das seqüências de microoperações descritas anteriormente. Por simplicidade, não são mostrados os caminhos de dados e de controle para incrementar o registrador PC e para carregar endereços fixos nos registradores PC e MAR.

Tabela 14.1 Microoperações e sinais de controle

Microoperações	Temporização	Sinais de controle ativos
Busca:	t1: MAR ← (PC)	C_2
	t2: MBR ← Memória	C_5, C_R
	PC ← (PC) + 1	
	t3: IR ← (MBR)	C_4
Indireto:	t1: MAR ← (IR(Endereço))	C_8
	t2: MBR ← Memória	C_5, C_R
	t3: IR(Endereço) ← (MBR(Endereço))	C_4
Interrupção:	t1: MBR ← (PC)	C_1
	t2: MAR ← Endereço de salvamento	
	PC ← Endereço de Rotina	
	t3: Memória ← (MBR)	C_{12}, C_W

C_R = sinal de controle de leitura para o barramento de sistema
C_W = sinal de controle de escrita para o barramento de sistema

Devemos notar que a unidade de controle apenas faz uso de elementos essenciais para desempenhar a sua função. Ela é o dispositivo que controla todo o computador, e o faz apenas com base no conhecimento sobre a próxima instrução a ser executada e sobre a natureza dos resultados de operações lógicas e aritméticas (por exemplo, valor positivo, *overflow* etc.). A unidade de controle nunca examina dados que estão sendo processados ou resultados realmente produzidos. Ela controla tudo enviando alguns sinais de controle para pontos dentro do processador e para o barramento de sistema.

Organização interna do processador

A Figura 14.5 mostra o uso de vários caminhos de dados. A complexidade desse tipo de organização deve ficar clara. Mais tipicamente, é usada alguma forma de arranjo de barramento interno, como sugerido na Figura 11.2.

Usando um barramento interno ao processador, a Figura 14.5 pode ser arranjada como mostrado na Figura 14.6. A ULA e todos os registradores do processador são conectados por um único barramento interno. São disponibilizadas portas e sinais de controle para a movimentação de dados entre o barramento e os registradores. Sinais de controle adicionais controlam a transferência de dados de e para o barramento (externo) de sistema, assim como a operação da ULA.

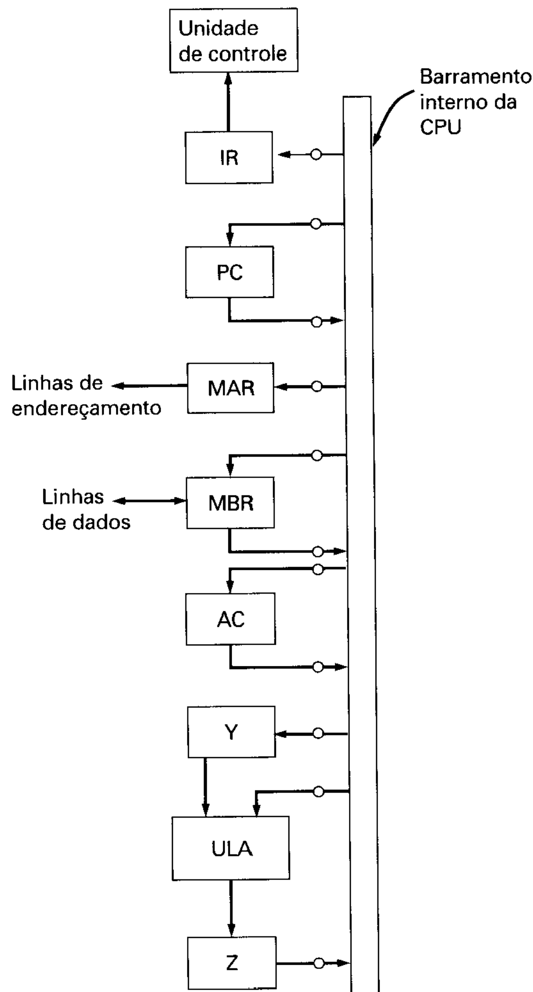


Figura 14.6 CPU com barramento interno.

São adicionados à organização dois novos registradores, designados como Y e Z. Esses registradores são necessários para que a ULA opere de forma apropriada. Quando é realizada uma operação envolvendo dois operandos, um deles pode ser obtido do barramento interno, mas o outro deve ser obtido de alguma outra fonte. O registrador acumulador (AC) poderia ser usado para esse propósito, mas isso limitaria a flexibilidade do sistema e não possibilitaria trabalhar com um processador que incluísse múltiplos registradores de propósito geral. O registrador Y oferece uma memória de armazenamento temporário para o outro dado de entrada. A ULA é um circuito combinatório (veja Apêndice A) que não possui memória interna. Portanto, quando sinais de controle ativam uma função da ULA, sua entrada é transformada em uma saída. A saída da ULA não pode ser diretamente conectada ao barramento porque essa saída seria redirecionada novamente como entrada. O registrador Z provê uma memória para armazenamento temporário da saída. Com esse arranjo, uma operação para somar um valor armazenado na memória ao registrador AC teria os seguintes passos:

t_1: MAR ← (IR(Endereço))
t_2: MBR ← Memória
t_3: Y ← (MBR)
t_4: Z ← (AC) + (Y)
t_5: AC ← (Z)

Embora sejam possíveis outras formas de organização, em geral é usada alguma forma de barramento interno ou conjunto de barramentos internos. O uso de caminhos de dados comuns simplifica o esquema de interconexão e o controle do processador. Outra razão prática para o uso de um barramento interno é a economia de espaço. Especialmente em microprocessadores, que podem ocupar área de apenas 1,6 centímetros quadrados de silício, o espaço ocupado por conexões entre registradores deve ser minimizado.

O Intel 8085

Para ilustrar alguns dos conceitos introduzidos até aqui, vamos examinar o processador Intel 8085. Sua organização é ilustrada na Figura 14.7. Alguns componentes principais que podem não ser diretamente compreendidos são:

- ***Latch* de incremento/decremento de endereço**: lógica que pode somar 1 ou subtrair 1 ao conteúdo do contador de programa ou do apontador de topo da pilha. Isso economiza tempo, evitando o uso da ULA para esse propósito.
- **Controle de interrupção**: esse módulo trata múltiplos níveis de sinais de interrupção.
- **Controle de E/S serial**: esse módulo faz a interface com dispositivos de comunicação serial, ou seja, dispositivos que comunicam 1 bit de cada vez.

A Tabela 14.2 descreve os sinais externos de entrada e de saída do processador 8085. Esses sinais são ligados ao barramento externo de sistema. Eles constituem a interface entre o processador 8085 e o restante do sistema (Figura 14.8).

A unidade de controle tem dois componentes identificados como (1) decodificador de instrução e codificação de ciclo de máquina e (2) temporização e controle. O primeiro componente é discutido na próxima seção. A essência da unidade de controle é o módulo de temporização e controle. Esse módulo inclui um relógio e tem como entradas a instrução corrente e alguns sinais de controle externos. Sua saída consiste de sinais de controle para os demais componentes do processador e de sinais para o barramento externo do sistema.

A temporização de operações do processador é feita por meio do relógio e controlada pela unidade de controle por meio de sinais de controle. Cada ciclo de instrução é dividido em um a cinco *ciclos de máquina*; cada ciclo de máquina, por sua vez, é dividido em três a cinco *estados*. Cada estado dura um ciclo de relógio. Durante um estado, o processador executa uma microoperação ou um conjunto de microoperações simultâneas, conforme determinado pelos sinais de controle.

O número de ciclos de máquina para uma dada instrução é fixo, mas esse número varia de uma instrução para outra. Os ciclos de máquina são definidos como sendo equivalentes a acessos ao barramento. Portanto, o número de ciclos de máquina para uma instrução depende do número de vezes que o processador tem de se comunicar com dispositivos externos. Por exemplo, se uma instrução consiste de duas partes de 8 bits, são requeridos dois ciclos de máquina para buscar a instrução. Se essa instrução envolve uma operação sobre 1 byte da memória ou de um dispositivo de E/S, é requerido um terceiro ciclo de máquina para a sua execução.

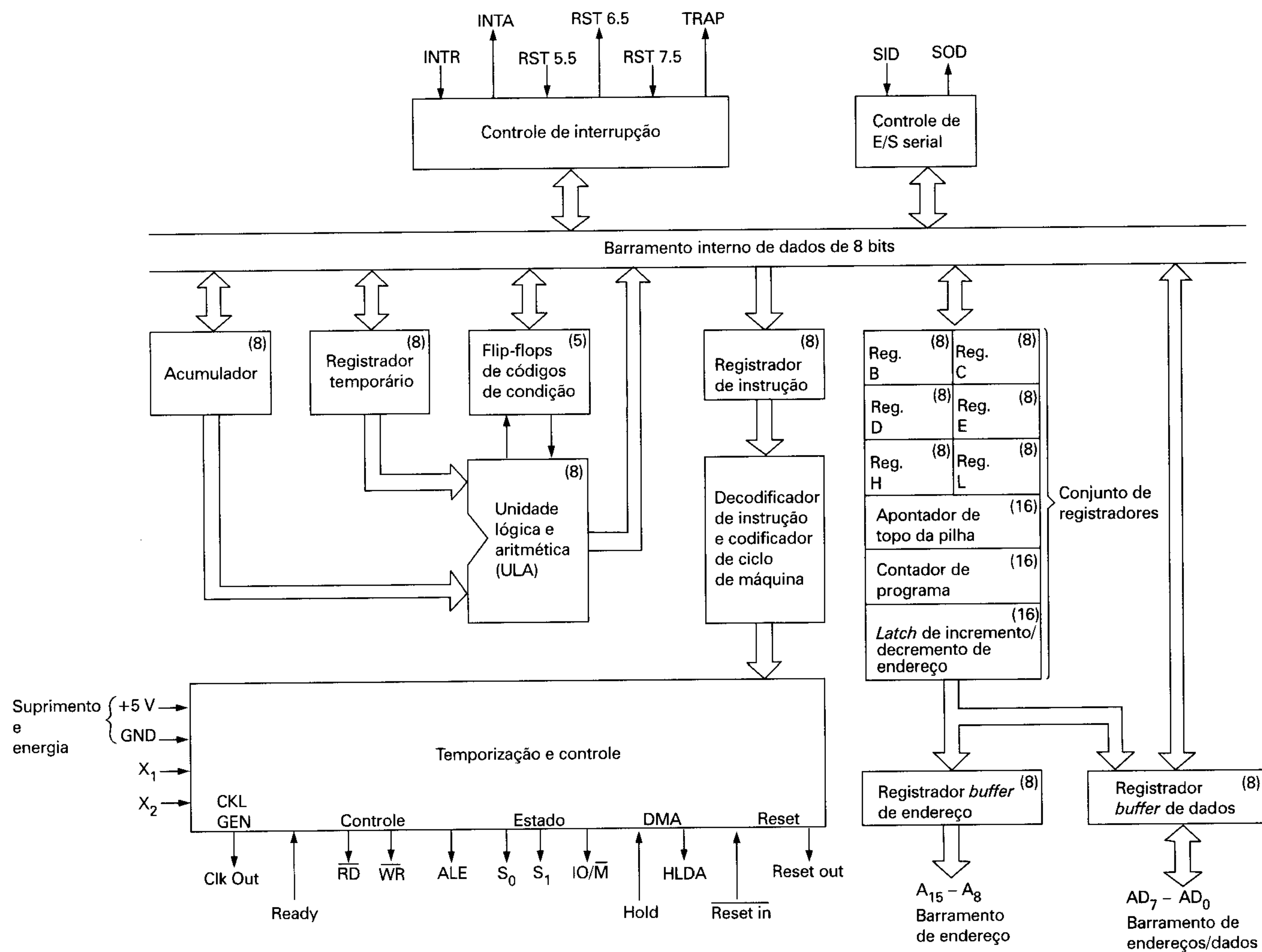


Figura 14.7 Diagrama de blocos do processador Intel 8085.

Tabela 14.2 Sinais externos do Intel 8085

Sinais de dados e de endereço

Endereço mais alto (A_{15}–A_8)

Os 8 bits de mais alta ordem de um endereço de 16 bits.

Endereço mais baixo/Dados (AD_7–AD_0)

Os 8 bits de mais baixa ordem de um endereço de 16 bits ou 8 bits de dados. Essa multiplexação economiza pinos de conexão.

Dados de entrada serial (SID)

Entrada de um bit para acomodar dispositivos que transmitem serialmente (1 bit por vez).

Dados de saída serial (SOD)

Saída de um bit para acomodar dispositivos que recebem serialmente.

Sinais de temporização e de controle

CLK (OUT)

Relógio do sistema. Cada ciclo representa um estado T. O sinal CLK vai para as pastilhas periféricas e sincroniza o seu funcionamento.

X1, X2

Esses sinais vêm de um cristal externo ou outro dispositivo que controla o gerador de relógio interno.

Habilitação de chave de endereço (ALE)

Ocorre durante o primeiro estado do relógio de um ciclo de máquina e faz com que as pastilhas periféricas armazenem as linhas de endereço. Isso possibilita que o módulo com endereço (por exemplo, memória, E/S) reconheça que está sendo endereçado.

Estado (S0, S1)

Sinais de controle usados para indicar se está em curso uma operação de leitura ou de escrita.

IO/ $\overline{M}$

Usado para habilitar um módulo de memória ou de E/S para operações de leitura ou de escrita.

Controle de leitura (RD)

Indica que o módulo de memória ou de E/S selecionado deve ser lido e que o barramento de dados está disponível para transferência.

Controle de escrita (WR)

Indica que o dado no barramento de dados deve ser escrito na posição de memória ou de E/S selecionada.

Sinais iniciados pela memória ou pela E/S

Parada (HOLD)

Requisita à CPU para retomar o controle e usar o barramento externo do sistema. A CPU completará a execução da instrução corrente no IR e então entrará em estado de parada, durante o qual nenhum sinal será inserido pela CPU nos barramentos de dados, endereço e controle. Durante o estado de espera, o barramento pode ser usado para operações de DMA.

Reconhecimento de parada (HOLDA)

Esse sinal de saída da unidade de controle reconhece o sinal HOLD e indica que o barramento está agora disponível.

Pronto (READY)

Usado para sincronizar a CPU com memórias e dispositivos de E/S mais lentos. Quando o dispositivo endereçado ativa o sinal READY, a CPU pode prosseguir com uma operação de entrada (DBIN) ou de saída (WR). Caso contrário, a CPU entra em um estado de espera até que o dispositivo esteja disponível.

Sinais relacionados a interrupção

TRAP

Interrupções de reinício (RST 7.5, 6.5, 5.5)

Tabela 14.2 Sinais externos do Intel 8085 (*continuação*)

Requisição de interrupção (INTR)	
	Essas cinco linhas são usadas por um dispositivo externo para interromper a CPU. A CPU não atenderá à requisição se ela estiver em estado de parada ou se a interrupção estiver desabilitada. A interrupção será atendida apenas quando a instrução for completada. As interrupções têm ordem de prioridade descendente.
Reconhecimento de interrupção	
	Reconhece uma interrupção.
Inicialização da CPU	
Reinício (RESET IN)	
	Faz com que o conteúdo do PC seja atualizado para zero. A CPU retoma a execução a partir da posição de endereço zero.
Reconhece reinício (RESET OUT)	
	Reconhece que a CPU foi reiniciada. Esse sinal pode ser usado para reiniciar o resto do sistema.
Voltagem e terra	
VCC	
	Suprimento de energia de +5 volts.
VSS ou GND	
	Terra elétrico.

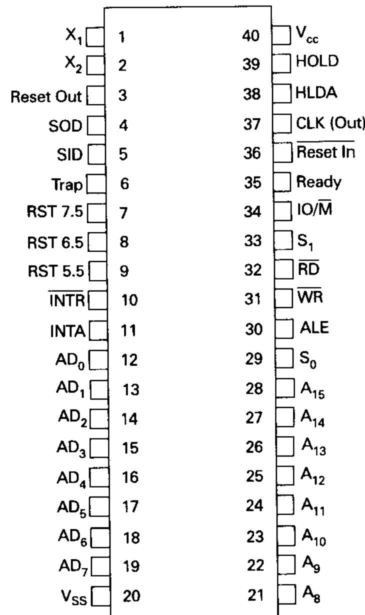


Figura 14.8 Configuração dos pinos do Intel 8085.

A Figura 14.9 apresenta um exemplo de temporização da execução de uma instrução do 8085, mostrando o valor de sinais de controle externos. É claro que, ao mesmo tempo, a unidade de controle gera também sinais de controle internos, para controlar as transferências de dados internas ao processador. O diagrama mostra o ciclo de instrução para uma instrução OUT. São requeridos três ciclos de máquina (M_1, M_2, M_3).

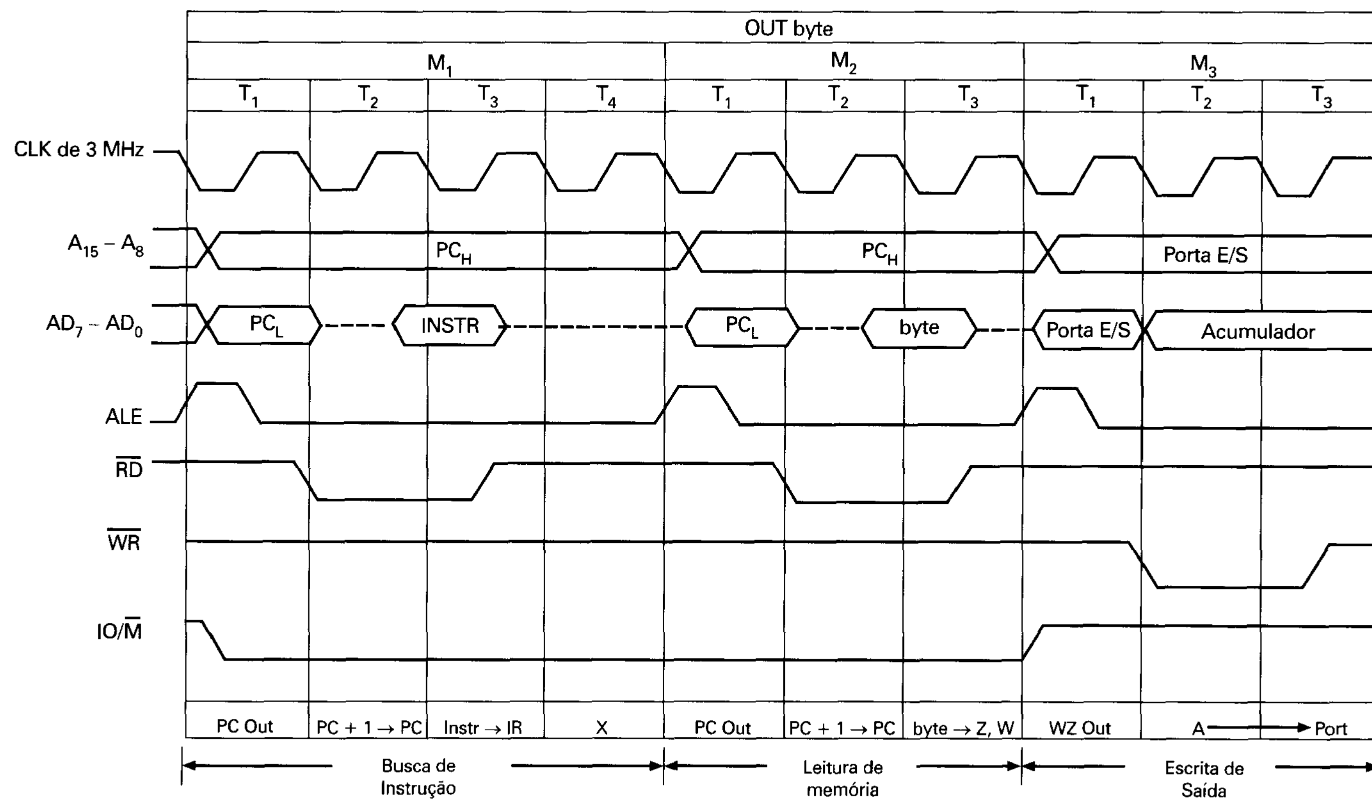


Figura 14.9 Diagrama de tempo para a instrução OUT do Intel 8085.

Durante o primeiro ciclo de máquina, a instrução OUT é buscada. O segundo ciclo busca a segunda metade da instrução, que contém o número do dispositivo de E/S selecionado para a saída. Durante o terceiro ciclo, o conteúdo do acumulador, AC, é escrito no dispositivo selecionado por meio do barramento de dados.

O início de cada ciclo é sinalizado pela unidade de controle pulsando o sinal de Habilitação de Latch de Endereço (ALE). O pulso do sinal ALE alerta circuitos externos. Durante o estado T_1 do ciclo de máquina M_1, a unidade de controle ativa o sinal IO/M, para indicar que se trata de uma operação de memória. Além disso, a unidade de controle faz com que o conteúdo de PC seja colocado no barramento de endereço ($A_{15} - A_8$) e no barramento de endereço/dados ($AD_7 - AD_0$). Durante a borda de descida do pulso ALE, os demais módulos conectados ao barramento armazenam o endereço.

Durante o estado T_2, o módulo de memória endereçado coloca o conteúdo da posição de memória endereçada no barramento de endereço/dados. A unidade de controle ativa o sinal de Controle de Leitura (RD), para indicar uma leitura, mas espera até o estado T_3 para copiar o dado do barramento. Isso dá tempo para que o módulo de memória coloque o dado no barramento e para que o nível do sinal se estabilize. O estado final, T_4, é um estado de *barramento ocioso*, durante o qual o processador decodifica a instrução. Os demais ciclos de máquina procedem de forma similar.

14.3 IMPLEMENTAÇÃO POR HARDWARE

Até agora, discutimos a unidade de controle em termos de suas entradas, saídas e funções. Está na hora de abordarmos a implementação da unidade de controle. Uma grande variedade de técnicas tem sido usada, e a maioria delas cai em uma das duas seguintes categorias:

- Implementação por hardware
- Implementação microprogramada

Em uma implementação por hardware, a unidade de controle é essencialmente um circuito combinatório. Seus sinais lógicos de entrada são transformados em um conjunto de sinais lógicos de saída, que constituem os sinais de controle. Essa abordagem é examinada nesta seção. A implementação microprogramada é assunto do Capítulo 15.

Entradas da unidade de controle

A Figura 14.4 mostra a unidade de controle tal como foi descrita até agora. As entradas básicas são o registrador de instrução, o relógio, os códigos de condição e os sinais do barramento de controle. No caso dos códigos de condição e dos sinais do barramento de controle, cada bit individual tipicamente tem um significado (por exemplo, *overflow*). As duas outras entradas, entretanto, não são usadas diretamente pela unidade de controle.

Considere primeiro o registrador de instrução. A unidade de controle usa o código de operação e executa ações diferentes (gera uma combinação diferente de sinais de controle) para diferentes instruções. Para simplificar a lógica da unidade de controle, deve existir uma única entrada lógica correspondente a cada código. Um *decodificador* é usado para ler uma entrada codificada e produzir uma saída única para cada código de operação. Em geral, o decodificador possui n entradas binárias e 2^n saídas binárias. Cada um desses 2^n diferentes padrões de bits de entrada ativa apenas uma única saída. A Tabela 14.3 mostra um exemplo.

O decodificador para uma unidade de controle deve ser tipicamente mais complexo que este, para levar em conta os códigos de operação de comprimento variável. Um exemplo da lógica digital usada para implementar um decodificador é apresentado no Apêndice A.

A parte do relógio da unidade de controle envia uma seqüência repetitiva de pulsos. Isso é útil para medir a duração das microoperações. Essencialmente, o período dos pulsos de relógio deve ser longo o suficiente para permitir a propagação dos sinais ao longo dos caminhos dos dados e por meio dos circuitos do processador. Contudo, como vimos até aqui, a unidade de controle envia diferentes sinais de controle em diferentes unidades de tempo, dentro de um único ciclo de instrução. Portanto, é necessário ter um contador como entrada para a unidade de controle, com um sinal de controle distinto para indicar intervalos de tempo T_1, T_2, e assim por diante. Ao final do ciclo de instrução, a unidade de controle deve reiniciar o contador em T_1.

Com esses dois refinamentos, a unidade de controle pode ser ilustrada tal como mostrado na Figura 14.10.

Lógica da unidade de controle

Para definir a implementação de uma unidade de controle por hardware, resta agora discutir a lógica interna da unidade de controle, que produz sinais de controle de saída como função dos seus sinais de entrada.

Essencialmente, é preciso derivar, para cada sinal de controle, uma expressão booleana que define esse sinal em função das entradas. Isso pode ser mais bem explicado por meio de um exemplo. Consideremos novamente o exemplo simples ilustrado na Figura 14.5. Vimos, na Tabela 14.1, as seqüências de microoperações e sinais de controle requeridos para controlar três das quatro fases do ciclo de instrução.

Tabela 14.3 Um decodificador com quatro entradas e dezesseis saídas

I1	I2	I3	I4	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Considere um único sinal de controle, C_5. Esse sinal faz com que seja lido um dado do barramento externo para o registrador MBR. Podemos ver que ele é usado duas vezes na Tabela 14.1. Vamos definir dois novos sinais de controle, P e Q, que têm a seguinte interpretação:

PQ = 00	Ciclo de busca
PQ = 01	Ciclo indireto
PQ = 10	Ciclo de execução
PQ = 11	Ciclo de interrupção

Então, a seguinte expressão booleana define C_5:

$$C_5 = \overline{P} \bullet \overline{Q} \bullet T_2 + \overline{P} \bullet Q \bullet T_2$$

Isto é, o sinal de controle C_5 é ativado durante o segundo intervalo de tempo, tanto do ciclo de busca como do ciclo indireto.

Essa expressão não está completa. C_5 também é necessário durante o ciclo de execução. No nosso exemplo simples, suponhamos que existam apenas três instruções que efetuem leitura na memória: LDA, ADD e AND. Podemos então definir C_5 como:

$$C_5 = \overline{P} \bullet \overline{Q} \bullet T_2 + \overline{P} \bullet Q \bullet T_2 + P \bullet \overline{Q} \bullet (LDA + ADD + AND) \bullet T_2$$

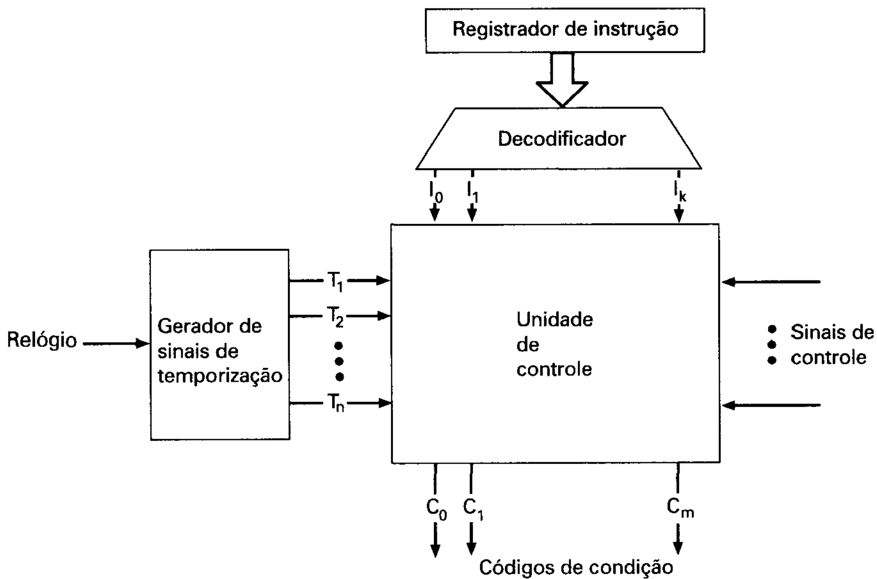


Figura 14.10 Unidade de controle com entradas decodificadas.

Esse mesmo processo pode ser repetido para todo sinal de controle gerado pelo processador. O resultado é um conjunto de equações booleanas que definem o comportamento da unidade de controle e, portanto, do processador.

Para juntar tudo, a unidade de controle deve controlar o estado do ciclo de instrução. Como mencionamos, ao fim de cada subciclo (busca, indireto, execução, interrupção), a unidade de controle emite um sinal que faz com que o gerador de sinais de temporização seja

reiniciado, e gere um sinal T_1. A unidade de controle deve também atribuir valores apropriados a P e Q para definir o próximo subciclo a ser efetuado.

Você pode imaginar que o número de equações booleanas requeridas para definir a unidade de controle em um complexo processador moderno é muito grande. A tarefa de implementar um circuito combinatório que satisfaça essas equações torna-se então extremamente difícil. O resultado é que usualmente é adotada uma abordagem muito mais simples, denominada *microprogramação*. Esse é o tema do próximo capítulo.

14.4 LEITURAS RECOMENDADAS

Diversos livros-texto discutem os princípios básicos de funcionamento da unidade de controle, incluindo Ward e Halstead (1990) e Mano e Kime (1997).

14.5 EXERCÍCIOS

14.1 Uma ULA pode somar seus dois registradores de entrada e pode complementar logicamente os bits de cada registrador de entrada, mas não pode subtrair. Os números são armazenados na representação de complemento de dois. Liste as microoperações que a unidade de controle deve executar para efetuar uma subtração.

14.2 Mostre as microoperações e os sinais de controle, tal como na Tabela 14.1, para que o processador da Figura 14.5 execute as seguintes instruções:

- Carregar acumulador
- Armazenar na memória o conteúdo do acumulador
- Somar ao acumulador
- Efetuar AND com o acumulador
- Desviar incondicionalmente
- Desviar se AC = 0
- Complementar o acumulador

14.3 Suponha que o tempo de atraso de propagação de sinais através do barramento e através da ULA da Figura 14.6 são 20 ns e 100 ns, respectivamente. O tempo requerido para que um registrador copie dados do barramento é 10 ns. Qual é o tempo gasto para:

a. Transferir dados de um registrador para outro?
b. Incrementar o contador de programa?

14.4 Escreva a seqüência de microoperações requerida para que a estrutura de barramento da Figura 14.6 some um número ao acumulador, quando esse número é:

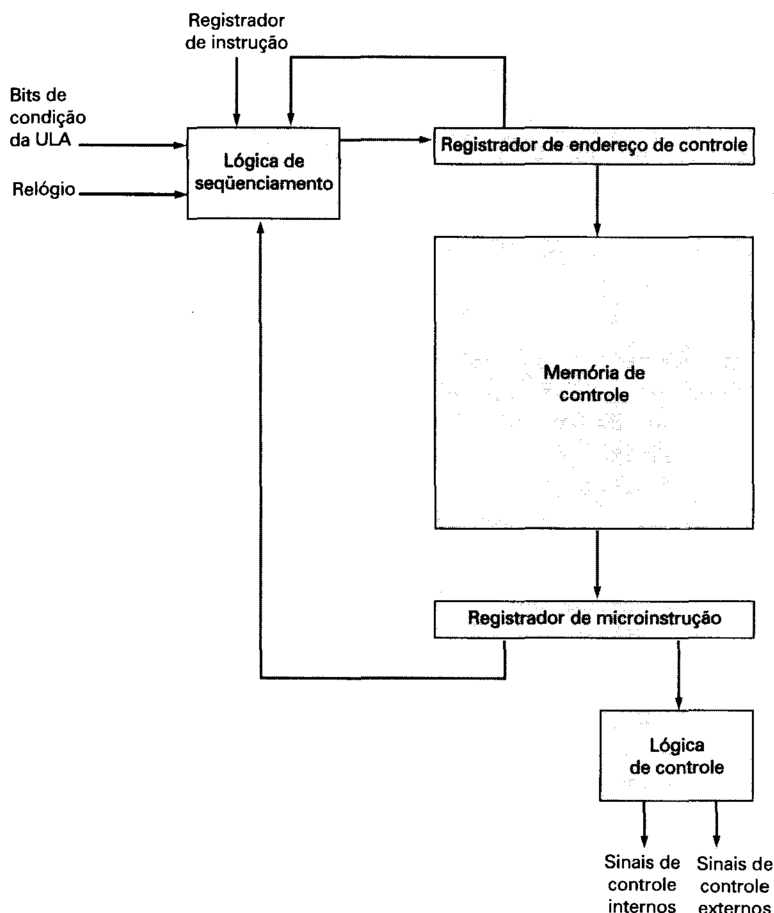
a. Um operando imediato
b. Um operando endereçado diretamente
c. Um operando endereçado indiretamente

14.5 Uma pilha é implementada como mostrado na Figura 9.14. Mostre a seqüência de microoperações para:

a. Desempilhar
b. Empilhar

capítulo

15 CONTROLE MICROPROGRAMADO



- Uma alternativa para a unidade de controle implementada em hardware é uma unidade de controle microprogramada, na qual a lógica da unidade de controle é especificada por um microprograma. Um microprograma consiste de uma seqüência de instruções de uma linguagem de microprogramação, que são instruções muito simples que especificam microoperações.
- Uma unidade de controle microprogramada tem um circuito lógico relativamente simples que é capaz de: (1) seguir uma seqüência de microinstruções e (2) gerar sinais de controle para executar cada microinstrução.
- Assim como em uma unidade de controle implementada em hardware, os sinais de controle gerados por uma microinstrução são usados para causar transferências de dados para registradores e execução de operações pela ULA.

O termo *microprograma* foi usado pela primeira vez por M. V. Wilkes no início da década de 50 (Wilkes, 1951). Wilkes propôs uma abordagem para o projeto de unidades de controle de forma organizada e sistemática e que evitava a complexidade das implementações por hardware. A idéia intrigou muitos pesquisadores, mas parecia difícil de ser aplicada, porque requeria uma memória de controle rápida e relativamente barata.

Uma revisão sobre o estado da arte na área de microprogramação foi apresentada na edição da revista *Datamation* de fevereiro de 1964. Nenhum sistema microprogramado era amplamente usado nessa época, e um dos artigos (Hill, 1964) expressava a então popular opinião de que o futuro da microprogramação era "um pouco nebuloso. Nenhum dos maiores fabricantes tem mostrado interesse nessa técnica, embora provavelmente todos a tenham examinado".

Essa situação mudou dramaticamente poucos meses depois. O Sistema/360 da IBM foi anunciado em abril e todos os modelos da série, exceto os de maior porte, eram microprogramados. Embora a série 360 seja anterior ao aparecimento da memória ROM de semicondutor, as vantagens da microprogramação eram suficientemente atrativas para fazer com que a IBM adotasse essa abordagem. Desde então, a microprogramação tornou-se um veículo cada vez mais popular para uma variedade de aplicações, uma das quais é o uso de microprogramação para implementar a unidade de controle de um processador. Essa aplicação é examinada neste capítulo.

15.1 CONCEITOS BÁSICOS

Microinstruções

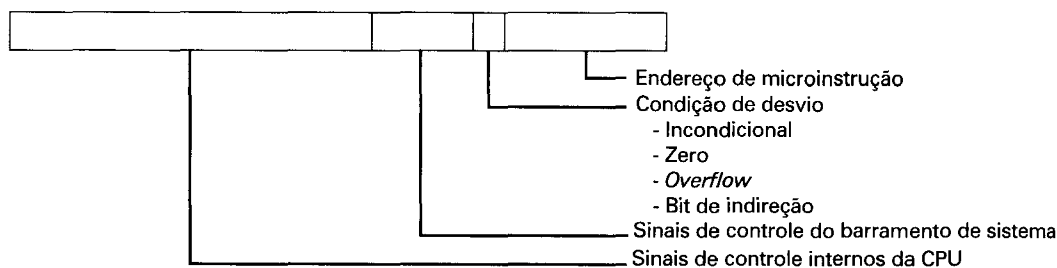
A unidade de controle parece ser um dispositivo razoavelmente simples. Entretanto, a implementação de uma unidade de controle como uma interconexão de elementos lógicos básicos não é uma tarefa fácil. O projeto deve incluir uma lógica para seqüenciamento por meio de microoperações, para executar as microoperações, para interpretar códigos de operação e para tomar decisões baseadas em códigos de condição da ULA. É difícil projetar e testar tal circuito de hardware. Além disso, o projeto é relativamente pouco flexível. Por exemplo, é difícil alterar o projeto para incluir uma nova instrução de máquina.

Uma alternativa de projeto, que é bastante comum nos processadores CISC contemporâneos, é a implementação de uma unidade de controle microprogramada.

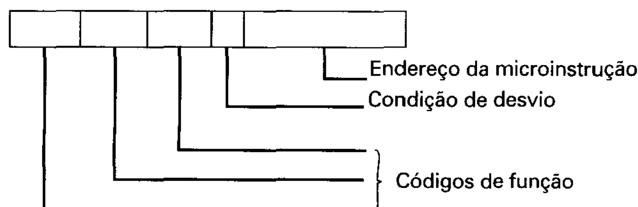
Considere novamente a Tabela 14.1. Além de usar sinais de controle, cada microoperação é descrita em notação simbólica. Essa notação se parece bastante com uma linguagem de programação. De fato, ela é uma linguagem, conhecida como *linguagem de microprogramação.* Cada linha descreve um conjunto de microoperações que ocorrem ao mesmo tempo e é conhecida como *microinstrução.* Uma seqüência de tais instruções é um *microprograma,* ou *firmware.* Esse último termo reflete o fato de que um microprograma é um meio-termo entre o hardware e o software. É mais fácil projetar um firmware do que um hardware, mas é mais difícil escrever um programa de firmware do que um programa de software.

Como o conceito de microprogramação pode ser usado para implementar a unidade de controle? Suponha que, para cada microoperação, tudo o que é permitido para a unidade de controle fazer seja gerar um conjunto de sinais. Então, para cada microoperação, cada linha de controle que sai da unidade de controle pode estar ativada ou desativada. Essas condições,

é claro, podem ser representadas como um valor binário 1 ou 0, ou seja, cada linha de controle pode ser representada por um dígito binário. Assim, podemos construir uma *palavra de controle*, onde cada bit representa uma linha de controle. Então, cada microoperação pode ser representada por um padrão distinto de 1s e 0s na palavra de controle.



(a) Microinstrução horizontal



(b) Microinstrução vertical

Figura 15.1 Formatos típicos de microinstrução.

Suponha que representemos uma seqüência de microoperações efetuadas pela unidade de controle como uma seqüência de palavras de controle. Devemos reconhecer, em seguida, que a seqüência de microoperações não é fixa. Algumas vezes temos um ciclo de indireção e outras vezes não. Coloquemos, então, nossas palavras de controle na memória, indicando a posição da próxima palavra de controle a ser executada caso uma determinada condição seja verdadeira (por exemplo, se o bit de indireção da instrução for igual a 1). Além disso, adicionemos alguns bits de controle para especificar a condição.

O resultado é conhecido como *microinstrução horizontal*. Um exemplo é apresentado na Figura 15.1a. O formato da microinstrução ou palavra de controle é o seguinte: existe um bit para cada linha de controle interna do processador e um bit para cada linha de controle do barramento de sistema. Um campo de condição indica a condição sob a qual deve ser efetuado um desvio, e outro campo indica o endereço da microinstrução a ser executada caso o desvio seja efetuado. Essa microinstrução é interpretada do seguinte modo:

1. Para executar essa microinstrução, ative todas as linhas de controle indicadas por um bit de valor 1, deixando inativas as linhas de controle indicadas por bits de valor 0. Os sinais de controle resultantes fazem com que uma ou mais microoperações sejam executadas.
2. Se a condição especificada no campo de condição for falsa, execute a próxima microinstrução da seqüência.

3. Se a condição especificada no campo de condição for verdadeira, a próxima microinstrução a ser executada será aquela indicada no campo de endereço.

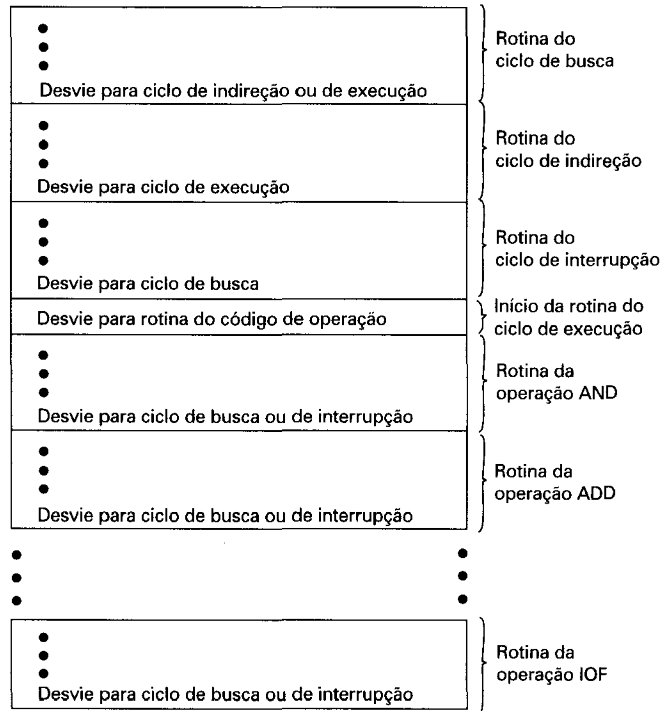


Figura 15.2 Organização da memória de controle.

A Figura 15.2 mostra como essas palavras de controle ou microinstruções podem ser arranjadas em uma *memória de controle*. As microinstruções de cada rotina são executadas seqüencialmente. Cada rotina termina com um desvio que indica para onde ir em seguida. Existe uma rotina especial do ciclo de execução que tem o único propósito de indicar qual das rotinas de instrução de máquina (AND, ADD, e assim por diante) deve ser executada em seguida, dependendo do código de operação corrente.

A memória de controle da Figura 15.2 é uma descrição concisa da operação completa da unidade de controle. Ela define a seqüência de microoperações a serem efetuadas durante cada ciclo (busca, indireção, execução, interrupção) e especifica a ordem na qual esses ciclos ocorrem. Essa notação constitui, no mínimo, uma forma útil de documentar o funcionamento da unidade de controle de um computador particular. Na verdade, ela é mais que isso: é também uma forma de implementar a unidade de controle.

Unidade de controle microprogramada

A memória de controle da Figura 15.2 contém um programa que descreve o comportamento da unidade de controle. Isso significa que podemos implementar a unidade de controle simplesmente executando esse programa.

A Figura 15.3 mostra os elementos-chave dessa implementação. O conjunto de microinstruções é armazenado na *memória de controle*. O *registrador de endereço de controle* contém o

endereço da próxima microinstrução a ser lida. Quando uma microinstrução é lida da memória de controle, ela é transferida para um *registrador de microinstrução* ou *registrador buffer de controle*. A porção mais à esquerda desse registrador (veja Figura 15.1a) é conectada às linhas de controle que saem da unidade de controle. Assim, *ler* uma microinstrução da memória de controle é o mesmo que *executar* essa microinstrução. O terceiro elemento mostrado na figura é uma unidade de seqüenciamento, que carrega o registrador de endereço de controle e envia um comando de leitura para a memória de controle.

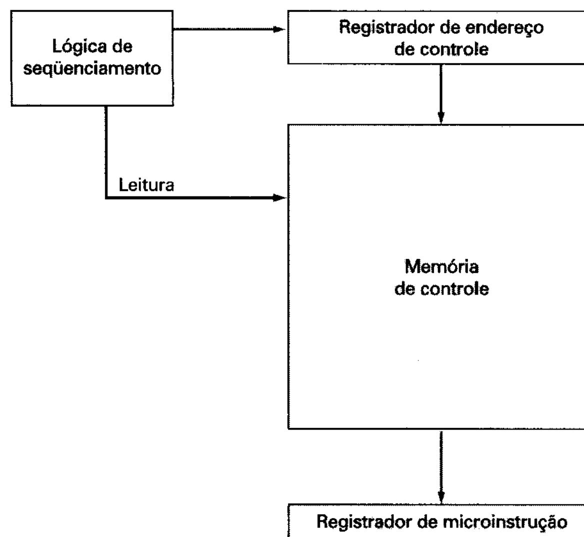


Figura 15.3 Microarquitetura da unidade de controle.

Examinemos essa estrutura mais detalhadamente, como mostrado na Figura 15.4. Comparando essa figura com a Figura 14.4, podemos ver que a unidade de controle ainda tem as mesmas entradas (IR, bits de condição da ULA, relógio) e as mesmas saídas (sinais de controle). A unidade de controle funciona do seguinte modo:

1. Para executar uma microinstrução, a unidade de lógica de seqüenciamento envia um comando de leitura para a memória de controle.
2. A palavra cujo endereço é especificado no registrador de endereço de controle é lida para o registrador de microinstrução.
3. O conteúdo desse registrador gera os sinais de controle e a informação sobre o próximo endereço para a unidade de lógica de seqüenciamento.
4. A unidade de lógica de seqüenciamento carrega o novo endereço no registrador de endereço de controle, com base na informação de próximo endereço obtida do registrador de microinstrução e nos bits de condição da ULA.

Tudo isso ocorre durante um ciclo de relógio.

O último passo acima requer uma explicação mais detalhada. Ao concluir cada microinstrução, a unidade de lógica de seqüenciamento carrega um novo endereço no registrador de endereço de controle. Dependendo do valor dos bits de condição da ULA, três decisões podem ser tomadas:

- **Buscar a próxima instrução**: adicionar 1 ao registrador de endereço de controle.
- **Desviar para uma nova rotina com base em uma microinstrução de desvio:** carregar campo de endereço do registrador de microinstrução no registrador de endereço de controle.
- **Desviar para uma rotina de instrução de máquina**: carregar o registrador de endereço de controle baseado no código de operação contido no registrador de instrução (IR).

A Figura 15.4 mostra dois módulos rotulados como *decodificadores*. O decodificador superior traduz o código de operação do registrador de instrução (IR) em um endereço na memória de controle. O decodificador inferior não é usado para microinstruções horizontais, mas sim para *microinstruções verticais* (Figura 15.1b). Como foi dito, em uma microinstrução horizontal cada bit do campo de controle é ligado a uma linha de controle. Em uma microinstrução vertical, é usado um código para cada ação a ser efetuada (por exemplo, MAR ←(PC)), e o decodificador traduz esse código em sinais de controle individuais. A vantagem de microinstruções verticais é que elas são mais compactas (menor número de bits) que microinstruções horizontais, ao custo de uma pequena lógica e tempo de atraso adicionais.

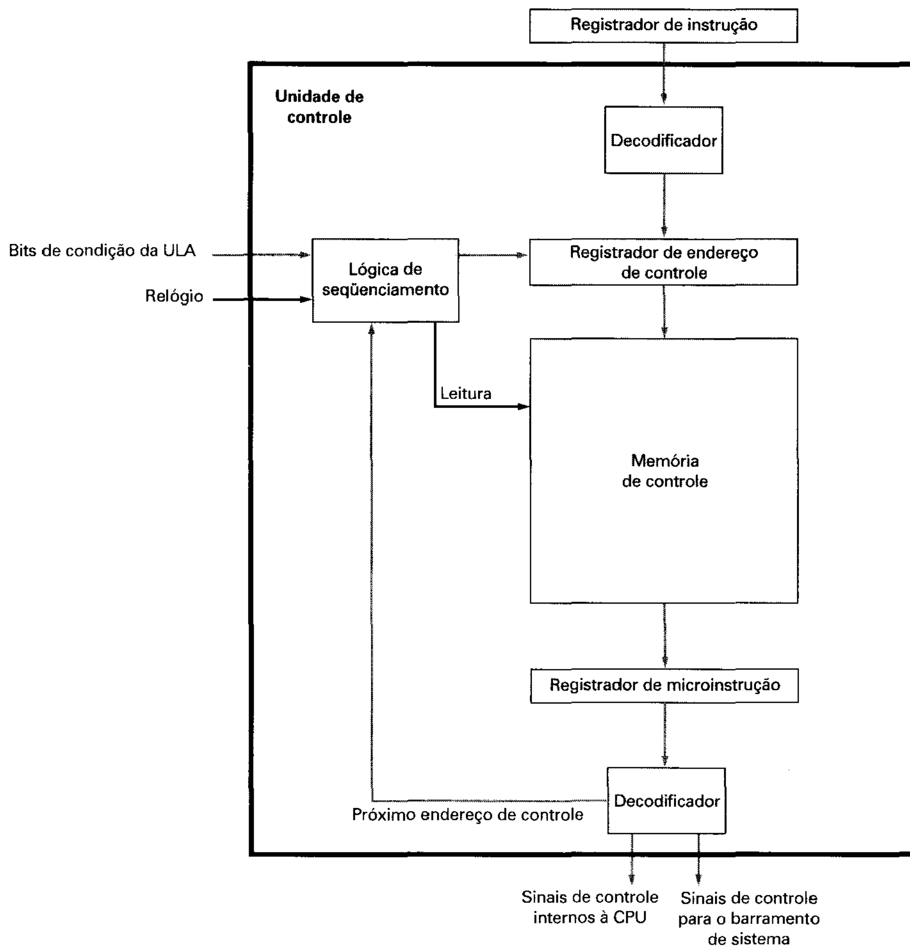


Figura 15.4 Funcionamento de uma unidade de controle microprogramada.

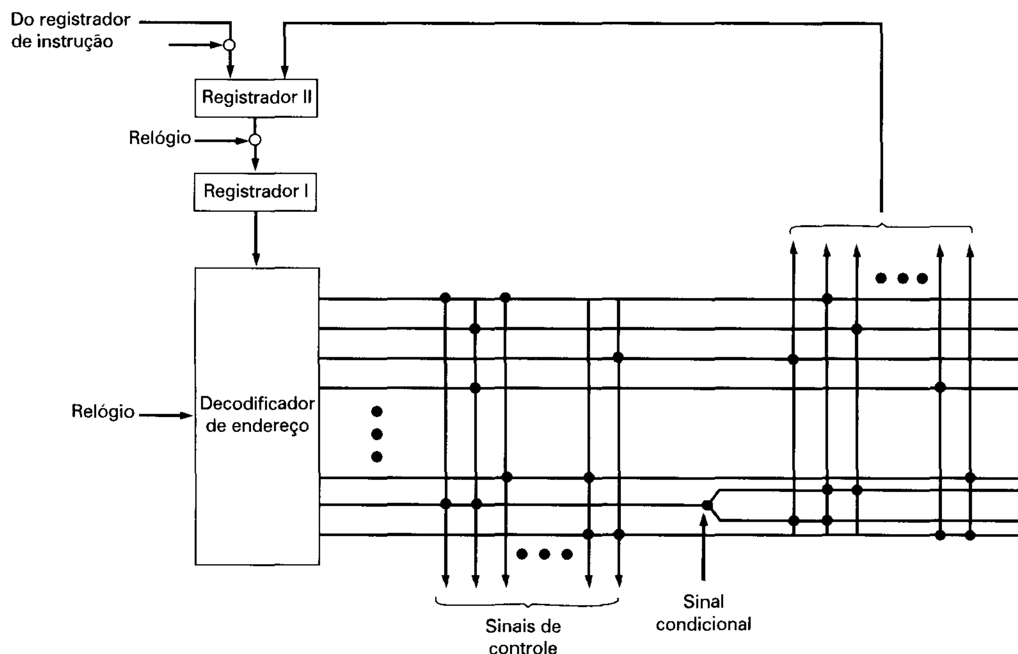


Figura 15.5 Unidade de controle microprogramada de Wilkes.

Controle de Wilkes

Como foi mencionado, Wilkes foi o primeiro a propor o uso de unidades de controle microprogramadas em 1951 (Wilkes, 1951). Essa proposta foi aperfeiçoada mais tarde em um projeto mais detalhado (Wilkes, 1953). É instrutivo examinar essa proposta original.

A configuração proposta por Wilkes é mostrada na Figura 15.5. O núcleo do sistema é uma matriz parcialmente preenchida com diodos. Durante um ciclo de máquina, uma linha da matriz é ativada com um pulso. Isso gera sinais nos pontos em que um diodo está presente (indicado por um ponto no diagrama). A primeira parte da linha gera sinais de controle que controlam a operação do processador. A segunda parte gera o endereço da linha a ser usada, para emissão de um pulso, no próximo ciclo de máquina. Portanto, cada linha da matriz é uma microinstrução e o leiaute da matriz é a memória de controle.

No início do ciclo, o endereço da linha a ser usada para emissão de um pulso está contido no Registrador I. Esse endereço é a entrada do decodificador, que, quando ativado por um pulso de relógio, ativa uma linha da matriz. Dependendo dos sinais de controle, ou o código de operação do registrador de instrução ou a segunda parte da linha usada para emissão do pulso é passado para o Registrador II durante o ciclo. O valor do Registrador II é, então, passado para o Registrador I por um pulso de relógio. São usados pulsos de relógio alternados para ativar uma linha da matriz e para transferir o conteúdo do Registrador II para o Registrador I. O arranjo de dois registradores é necessário porque o decodificador é, simplesmente, um circuito combinatório; com apenas um registrador, a saída seria redirecionada para a entrada durante o mesmo ciclo, causando uma condição de instabilidade.

Esse esquema é muito similar à abordagem de microprogramação horizontal descrita anteriormente (Figura 15.1a). A principal diferença é que, na descrição anterior, o registrador

de endereço de controle podia ser incrementado de um para se obter o próximo endereço. No esquema de Wilkes, o próximo endereço está contido nas microinstruções. Para permitir desvios, uma linha deve conter duas partes de endereço, controladas por um sinal condicional (por exemplo, bits de condição), como mostrado na figura.

Tendo proposto esse esquema, Wilkes forneceu um exemplo de como ele poderia ser usado para implementar a unidade de controle de uma máquina simples. É interessante reproduzir aqui esse exemplo, o primeiro exemplo conhecido de um processador microprogramado, porque ele ilustra muitos dos princípios de microprogramação contemporâneos.

O processador da máquina hipotética incluía os seguintes registradores:

A multiplicando
B acumulador (metade menos significativa)
C acumulador (metade mais significativa)
D registrador de deslocamento

Além desses, existem três registradores e um código de condição de 2 bits, acessíveis apenas à unidade de controle. Os registradores são os seguintes:

E serve como registrador de endereço de memória (MAR) ou como registrador de armazenamento temporário
F contador de programa
G outro registrador temporário; usado como contador

A Tabela 15.1 relaciona o conjunto de instruções de máquina desse exemplo. A Tabela 15.2 mostra o conjunto completo de microinstruções, expresso de forma simbólica, que implementa a unidade de controle. Ao todo, são requeridas apenas 38 microinstruções para definir completamente o sistema.

Tabela 15.1 Conjunto de instruções de máquina do exemplo de Wilkes

Instrução	Efeito da Instrução
$A\ n$	$C(Acc) + C(n)$ para Acc_1
$S\ n$	$C(Acc) - C(n)$ para Acc_1
$H\ n$	$C(n)$ para Acc_2
$V\ n$	$C(Acc_2) \times C(n)$ para Acc, onde $C(n) \geq 0$
$T\ n$	$C(Acc_1)$ para n, 0 para Acc
$U\ n$	$C(Acc_1)$ para n
$R\ n$	$C(Acc) \times 2^{-(n+1)}$ para Acc
$L\ n$	$C(Acc) \times 2^{n+1}$ para Acc
$G\ n$	Se $C(Acc) < 0$, transfere controle para n; se $C(Acc) \geq 0$, ignorar (prossegue serialmente)
$I\ n$	Lê o próximo caractere do mecanismo de entrada para n
$O\ n$	Envia $C(n)$ para o mecanismo de saída

Notação: Acc = acumulador
Acc_1 = metade mais significativa do acumulador
Acc_2 = metade menos significativa do acumulador
n = posição de memória de endereço n
$C(X)$ = conteúdo de X (X = registrador ou endereço de memória)

Tabela 15.2 Microinstruções do exemplo de Wilkes

Notação: *A*, *B*, *C*,... denotam os vários registradores da unidade aritmética e da unidade de controle. *C* para *D* indica que os circuitos de comutação conectam a saída do registrador *C* à entrada do registrador *D*; (*D* + *A*) para *C* indica que a saída do registrador *A* é conectada a uma entrada do somador (a saída de *D* é permanentemente conectada à outra entrada) e a saída do somador é conectada ao registrador *C*. Um símbolo numérico *n*, entre "aspas simples" (isto é, '*n*'), representa a fonte cuja saída é número *n* em unidades do dígito menos significativo.

	Unidade Aritmética	**Unidade de Registrador de controle**	**Flip-Flop Condicional**		**Próxima Microinstrução**	
			Ativação	Uso	0	1
0		*F* para *G* e *E*			1	
1		(*G* para '1') para *F*			2	
2		Armazenamento para *G*			3	
3		*G* para *E*			4	
4		*E* para decodificador			—	
A 5	*C* para *D*				16	
S 6	*C* para *D*				17	
H 7	Armazenamento para *B*				0	
V 8	Armazenamento para *A*				27	
T 9	*C* para Armazenamento				25	
U 10	*C* para Armazenamento				0	
R 11	*B* para *D*	*E* para *G*			19	
L 12	*C* para *D*	*E* para *G*			22	
G 13		*E* para *G*	(1)C_5		18	
I 14	Entrada para Armazenamento				0	
O 15	Armazenamento para Saída				0	
16	(*D* + Armazenamento) para *C*				0	
17	(*D* – Armazenamento) para *C*				0	
18				1	0	1
19	*D* para *B* (*R*)*	(*G* – '1') para *E*			20	
20	*C* para *D*		(1)E_5		21	
21	*D* para *C* (R)			1	11	0
22	*D* para *C* (*L*)⁺	(*G* – '1') para *E*			23	
23	*B* para *D*		(1)E_5		24	

Tabela 15.2 Microinstruções do exemplo de Wilkes (*continuação*)

24	D para B (L)			1	12	0
25	'0' para B				26	
26	B para C				0	
27	'0' para C	'18' para E			28	
28	B para D	E para G	$(1)B_1$		29	
29	D para B (R)	(G – '1') para E			30	
30	C para D (R)		$(2)E_5$	1	31	32
31	D para C			2	28	33
32	(D + A) para C			2	28	33
33	B para D		$(1)B_1$		34	
34	D para B (R)				35	
35	C para D (R)			1	36	37
36	D para C				0	
37	(D – A) para C				0	

* Deslocamento para a direita. Os circuitos de comutação da unidade aritmética são arranjados de forma que, durante uma microoperação de deslocamento para a direita, o dígito menos significativo do registrador *C* é colocado na posição mais significativa do registrador *B* e o dígito mais significativo do registrador *C* (dígito de sinal) é repetido (fazendo assim a correção para números negativos).

\+ Deslocamento para a esquerda. Os circuitos de comutação são arranjados de forma similar, para passar o dígito mais significativo do registrador *B* para a posição menos significativa do registrador *C* durante uma microoperação de deslocamento para a esquerda.

A primeira coluna cheia dá o endereço (número de linha) de cada microinstrução. Os endereços correspondentes a códigos de operação são rotulados. Assim, quando é encontrado o código de operação para a instrução de adição (A), a microinstrução localizada no endereço 5 é executada. As colunas 2 e 3 expressam as ações tomadas pela ULA e pela unidade de controle, respectivamente. Cada expressão simbólica deve ser traduzida em um conjunto de sinais de controle (bits de microinstrução). As colunas 4 e 5 estão relacionadas com a ativação e o uso dos dois bits de condição (flip-flops). A coluna 4 especifica o sinal que ativa o código de condição. Por exemplo, *(1)* C_s indica que o bit de condição 1 é ativado pelo bit de sinal do número contido no registrador C. Se a coluna 5 contém um identificador de bit de condição, então as colunas 6 e 7 contêm os dois endereços alternativos de microinstrução a serem usados. Caso contrário, a coluna 6 especifica o endereço da próxima microinstrução a ser buscada.

As instruções 0 a 4 constituem o ciclo de busca. A microinstrução 4 apresenta o código de operação para um decodificador, que gera o endereço da microinstrução correspondente à instrução de máquina a ser buscada. O leitor deve ser capaz de deduzir o completo funcionamento da unidade de controle a partir de um estudo meticuloso da Tabela 15.2.

Vantagens e desvantagens

A principal vantagem do uso de microprogramação para implementar uma unidade de controle é que ela simplifica o projeto da unidade de controle. Assim, sua implementação tanto é mais barata como é menos sujeita a erro. Uma unidade de controle implementada por hardware deve conter uma lógica complexa para seqüenciamento das diversas microoperações

do ciclo de instrução. Por outro lado, os decodificadores e a unidade de lógica de seqüenciamento de uma unidade de controle microprogramada possuem uma lógica muito simples.

A principal desvantagem de uma unidade de controle microprogramada é ser um pouco mais lenta que uma unidade de controle implementada em hardware de tecnologia comparável. Apesar disso, a microprogramação é a técnica dominante para implementação de unidades de controle de máquinas CISC contemporâneas, devido à sua facilidade de implementação. Os processadores RISC, com seu formato de instrução mais simples, tipicamente usam unidades de controle implementadas em hardware. A abordagem de microprogramação é examinada com mais detalhes a seguir.

15.2 SEQÜENCIAMENTO DE MICROINSTRUÇÕES

As duas tarefas básicas realizadas por uma unidade de controle microprogramada são:

- **Seqüenciamento de microinstruções**: buscar a próxima microinstrução da memória de controle.
- **Execução de microinstrução:** gerar os sinais de controle necessários para executar a microinstrução.

No projeto de uma unidade de controle, essas tarefas devem ser consideradas em conjunto, pois ambas afetam o formato das microinstruções e a temporização da unidade de controle. Nesta seção, enfocamos o seqüenciamento, abordando o menos possível questões sobre o formato e a temporização. Essas questões são tratadas mais detalhadamente na seção seguinte.

Considerações de projeto

O projeto de uma técnica de seqüenciamento de microinstruções envolve duas preocupações: o tamanho das microinstruções e o tempo de geração de endereços. A primeira preocupação é óbvia: minimizando o tamanho da memória de controle, o custo desse componente é reduzido. A segunda preocupação vem simplesmente do desejo de executar microinstruções tão rápido quanto possível.

Na execução de um microprograma, o endereço da próxima microinstrução a ser executada pode ser:

- Determinado pelo registrador de instrução
- O próximo endereço na seqüência
- Um endereço de desvio

A primeira categoria ocorre apenas uma vez por ciclo de instrução, exatamente depois que a instrução é buscada. A segunda categoria é a mais comum, na maioria dos projetos. Entretanto, o projeto não pode ser otimizado apenas para acesso seqüencial. Desvios, tanto condicionais como incondicionais, constituem parte necessária de um microprograma. Além disso, seqüências de microinstruções tendem a ser curtas; uma em cada três ou quatro microinstruções pode ser um desvio (Siewiorek, Bell e Newell, 1982). Portanto, é importante projetar técnicas de desvio para microinstruções que sejam compactas e eficientes no tempo.

Técnicas de seqüenciamento

Com base na microinstrução corrente, nos bits de condição e no conteúdo do registrador de instrução, deve ser gerado o endereço da próxima microinstrução na memória de controle. Uma grande variedade de técnicas tem sido usada. Essas técnicas podem ser agrupadas em

três categorias, conforme ilustrado nas Figuras 15.6 a 15.8. Essas categorias são baseadas no formato da informação de endereço na microinstrução:

- Dois campos de endereço
- Um único campo de endereço
- Formato variável

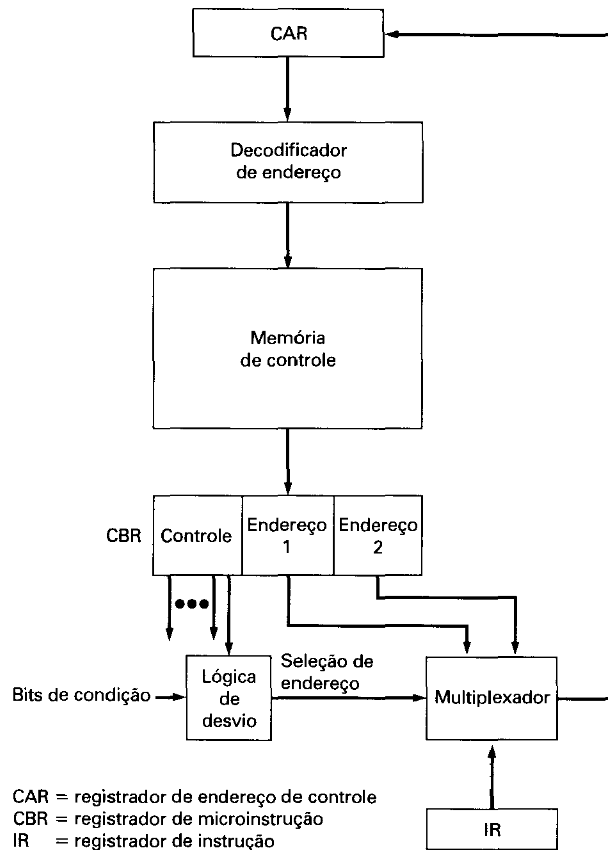


Figura 15.6 Lógica de controle de desvio com dois campos de endereço.

A abordagem mais simples é adotar dois campos de endereço em cada microinstrução. A Figura 15.6 sugere como essa informação é usada. Os dois campos de endereço e o registrador de instrução constituem as entradas para um multiplexador. Baseado nos sinais de seleção de endereço, o multiplexador transmite o código de operação ou um dos dois endereços para o registrador de endereço de controle (CAR). O registrador de endereço de controle é subseqüentemente decodificado, para produzir o endereço da próxima microinstrução. Os sinais de seleção de endereço são fornecidos por um módulo de lógica de desvio, cujas entradas consistem dos códigos de condição da unidade de controle e dos bits de controle da microinstrução.

Embora essa abordagem de dois endereços seja simples, ela requer mais bits na microinstrução que as demais abordagens. Com alguma lógica adicional, é possível economizar al-

guns bits. Uma abordagem comum é usar apenas um campo de endereço (Figura 15.7). Com essa abordagem, as opções para o próximo endereço são:

- O campo de endereço
- O código contido no registrador de instrução
- O próximo endereço na seqüência

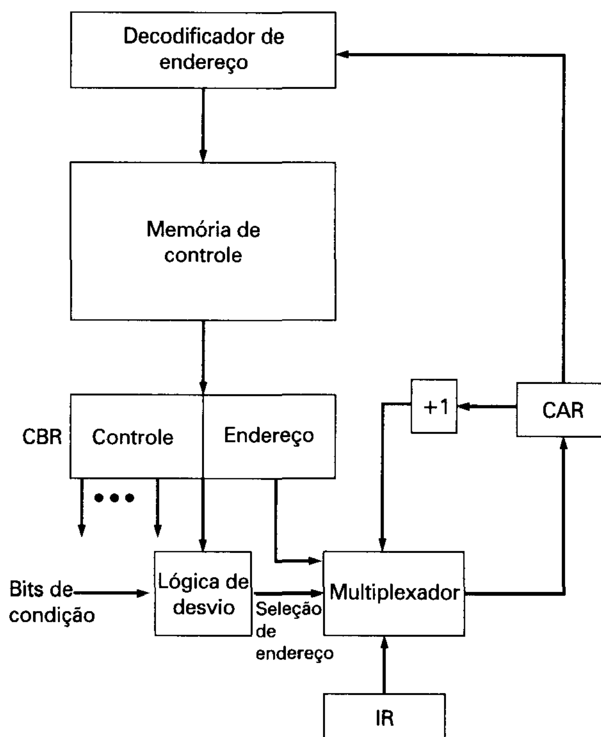


Figura 15.7 Lógica de controle de desvio com campo de endereço único.

Os sinais de seleção de endereço determinam que opção deve ser selecionada. Essa abordagem reduz para um o número de campos de endereço. Note, entretanto, que o campo de endereço freqüentemente não é usado. Portanto, existe ainda uma certa ineficiência no esquema de codificação de microinstruções.

Outra abordagem é possibilitar dois formatos de microinstrução inteiramente diferentes (Figura 15.8). Um bit designa o formato usado. Em um dos formatos, os demais bits são usados para ativar sinais de controle. No outro formato, alguns bits são usados para direcionar o módulo de controle de desvio e os bits restantes contêm o endereço. No primeiro formato, o endereço da próxima microinstrução é o próximo endereço seqüencial ou o endereço obtido do registrador de instrução. No segundo formato, é especificado um desvio condicional ou incondicional. Uma desvantagem dessa abordagem é consumir um ciclo completo em cada microinstrução de desvio. Nas outras duas abordagens, a geração de endereço ocorre como parte do mesmo ciclo em que os sinais de controle são gerados, minimizando acessos à memória de controle.

As abordagens descritas são genéricas. Implementações específicas freqüentemente envolvem uma variação ou combinação dessas técnicas.

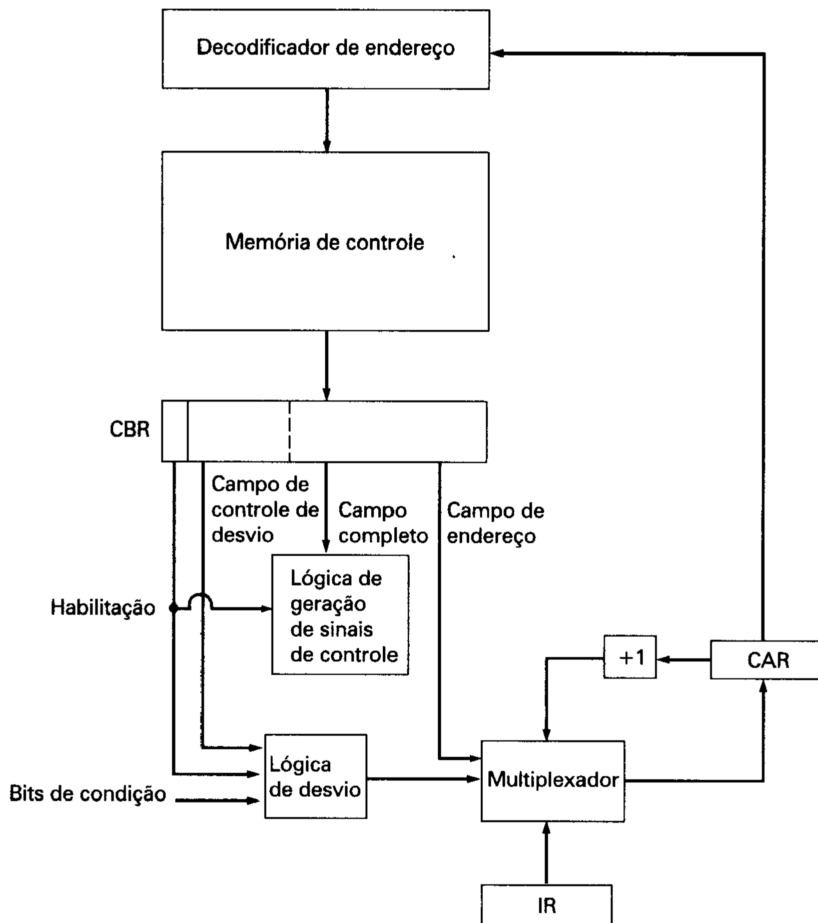


Figura 15.8 Lógica de controle de desvio com formato variável.

Geração de endereço

Vimos anteriormente o problema de seqüenciamento sob o ponto de vista de considerações de formato e dos requisitos de lógica gerais. Outro ponto de vista considera as várias formas para calcular o próximo endereço.

A Tabela 15.3 relaciona as várias técnicas de geração de endereço. Essas técnicas podem ser divididas em técnicas explícitas, nas quais o endereço está disponível explicitamente na microinstrução, e técnicas implícitas, que requerem uma lógica adicional para gerar o endereço.

Tabela 15.3 Técnicas de geração de endereço de microinstrução

Explícitas	**Implícitas**
Dois campos de endereço	Mapeamento
Desvio incondicional	Adição
Desvio condicional	Controle residual

Até agora, lidamos essencialmente com as técnicas explícitas. Na abordagem de dois campos de endereços, dois endereços alternativos estão disponíveis na microinstrução. Usando a abordagem de um endereço ou de formato variável, várias instruções de desvio podem ser implementadas. Uma instrução de desvio condicional depende dos seguintes tipos de informação:

- Bits de condição da ULA
- Parte do código de operação ou dos campos de modo de endereçamento da instrução de máquina
- Parte de um registrador selecionado, tal como o bit de sinal
- Bits de estado da unidade de controle

Diversas técnicas implícitas são também comumente usadas. Uma delas, o mapeamento, é requerida praticamente em todos os projetos. A porção correspondente ao código de operação de uma instrução de máquina deve ser mapeada em um endereço de microinstrução. Isso ocorre uma vez por ciclo de instrução.

Uma técnica implícita comum é a que envolve combinar ou somar duas porções de um endereço para formar o endereço completo. Essa abordagem foi adotada na família IBM S/360 (Tucker, 1987) e usada em muitos modelos do S/370. Vamos usar o IBM 3033 como exemplo.

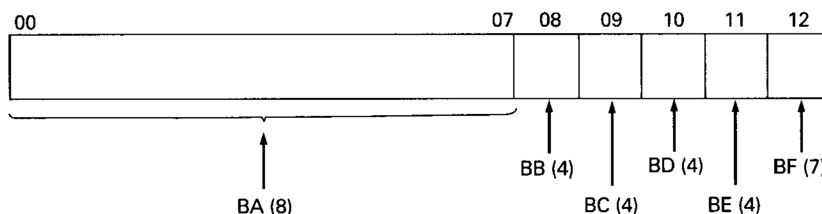


Figura 15.9 Registrador de endereço de controle do IBM 3033.

O registrador de endereço de controle do IBM 3033 tem 13 bits e é ilustrado na Figura 15.9. Duas partes do endereço são distinguidas. Os 8 bits de mais alta ordem (00-07) normalmente não mudam de um ciclo de microinstrução para o próximo. Durante a execução de uma microinstrução, esses bits são copiados diretamente de um campo de 8 bits da microinstrução (campo BA) para os 8 bits de mais alta ordem do registrador de endereço de controle. Isso define um bloco de 32 microinstruções na memória de controle. Os 5 bits restantes do registrador de endereço de controle são determinados de modo que especifiquem o endereço da próxima microinstrução a ser buscada. Cada um desses bits é determinado por um campo de 4 bits da microinstrução corrente (exceto um, que é determinado por um campo de 7 bits); o campo especifica a condição para atribuir valor 1 ao bit correspondente. Por exemplo, o valor de um bit do registrador de endereço de controle pode ser 1 ou 0, caso tenha ocorrido um 'vai-um' na última operação efetuada pela ULA.

A última abordagem mostrada na Tabela 15.3 é denominada *controle residual*. Essa abordagem envolve o uso de um endereço de microinstrução que tenha sido armazenado anteriormente em área de armazenamento temporário, dentro da unidade de controle. Por exemplo, alguns conjuntos de microinstruções incluem um mecanismo de sub-rotinas. Um registrador interno ou uma pilha de registradores é usado para armazenar o endereço de retorno. Essa abordagem é adotada, por exemplo, no LSI-11, examinado a seguir.

Seqüenciamento de microinstruções no LSI-11

O LSI-11 é uma versão de microcomputador de um PDP-11, com os principais componentes do sistema residindo em uma única placa. O LSI-11 foi implementado usando unidade de controle microprogramada (Sebern, 1976).

O LSI-11 usa microinstruções de 22 bits e uma memória de controle de 2 K palavras de 22 bits. O endereço da próxima microinstrução é determinado por uma das cinco maneiras seguintes:

- **Próximo endereço seqüencial**: na ausência de outras instruções, o registrador de endereço da unidade de controle é incrementado de 1.
- **Mapeamento de código de operação**: no início de cada ciclo de instrução, o endereço da próxima microinstrução é determinado pelo código de operação.
- **Mecanismo de sub-rotina**: explicado a seguir.
- **Teste de interrupção**: algumas microinstruções especificam um teste de interrupção. Se ocorrer uma interrupção, isso determina o endereço da próxima microinstrução.
- **Desvio**: são usadas microinstruções de desvio condicional e incondicional.

É definido um mecanismo de sub-rotina de um nível. Um bit em cada microinstrução é reservado para esse propósito. Quando esse bit tem valor 1, um registrador de retorno de 11 bits é carregado com o conteúdo atualizado do registrador de endereço de controle. Uma microinstrução subseqüente que especifique um retorno faz com que o registrador de endereço de controle seja carregado a partir do registrador de retorno.

O retorno é uma forma de instrução de desvio incondicional. Outra forma de desvio incondicional faz com que os bits do registrador de endereço de controle sejam carregados a partir dos 11 bits da microinstrução. A instrução de desvio condicional usa um código de teste de 4 bits da microinstrução. Esse código especifica o teste de vários códigos de condição da ULA, para decidir se o desvio deve ser tomado. Se a condição não for verdadeira, será selecionado o próximo endereço seqüencial. Se for verdadeira, os 8 bits de mais baixa ordem do registrador de endereço de controle serão carregados a partir de 8 bits da microinstrução. Isso possibilita desvios dentro de uma página de memória de 256 palavras.

Como se pode ver, o LSI-11 inclui um poderoso mecanismo de seqüenciamento de endereços dentro da unidade de controle. Isso permite ao microprogramador uma flexibilidade considerável e pode facilitar a tarefa de microprogramação. Por outro lado, essa abordagem requer mais lógica na unidade de controle do que abordagens mais simples.

15.3 EXECUÇÃO DE MICROINSTRUÇÕES

O ciclo de microinstrução é o evento básico de um processador microprogramado. Cada ciclo é constituído de duas partes: busca e execução. A porção de busca é determinada pela geração de um endereço de microinstrução e foi descrita na seção precedente. Esta seção trata da execução de microinstruções.

Lembre-se de que o efeito da execução de uma microinstrução é gerar sinais de controle. Alguns desses sinais controlam pontos internos do processador. Os sinais restantes vão para o barramento de controle ou outra interface externa. Além disso, é determinado o endereço da próxima microinstrução.

Essa descrição sugere a organização da unidade de controle ilustrada na Figura 15.10, que consiste em uma versão revisada da Figura 15.4, enfocada nesta seção. Os principais módulos desse diagrama já devem estar claros. O módulo de lógica de seqüenciamento contém a lógica para efetuar as funções descritas na seção precedente. Ele gera o endereço da próxima microinstrução, usando como entrada o registrador de instrução, os bits de condição da ULA, o registrador de endereço de controle (para obter o endereço da próxima instrução consecutiva) e o registrador de microinstrução. Esse último pode prover um endereço, bits de controle ou ambos. O módulo é dirigido por um relógio, que determina a temporização do ciclo de microinstrução.

O módulo de lógica de controle gera sinais de controle em função de alguns bits da microinstrução. Deve ficar claro que o formato e o conteúdo da microinstrução determinam a complexidade do módulo de lógica de controle.

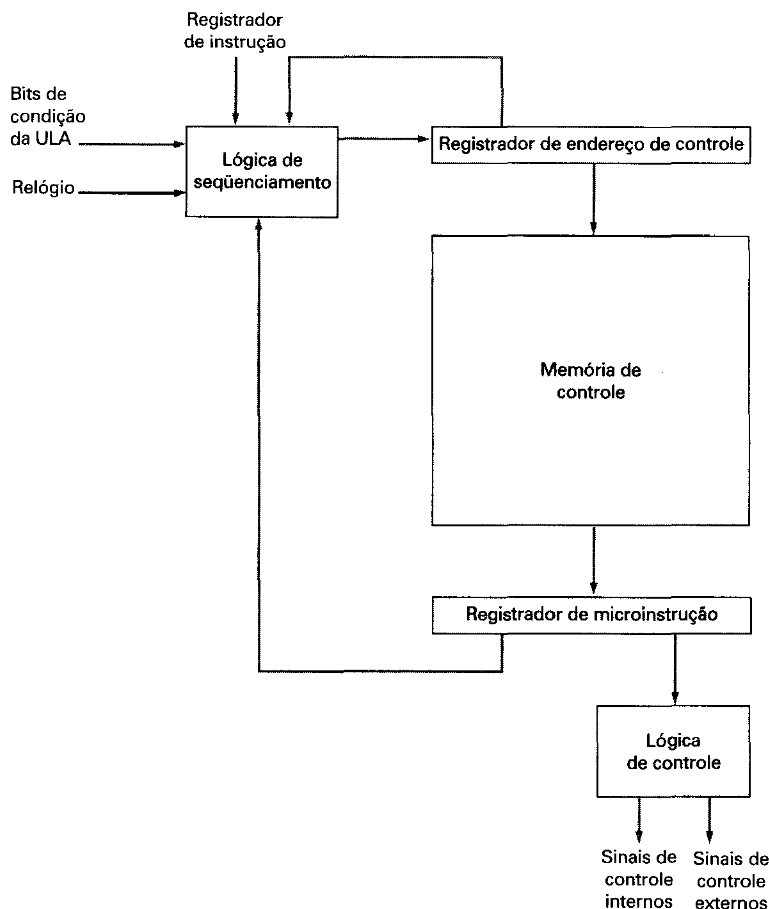


Figura 15.10 Organização da unidade de controle.

Uma taxonomia de microinstruções

Microinstruções podem ser classificadas de diversos modos. Distinções comumente encontradas na literatura incluem as seguintes:

- Vertical ou horizontal
- Empacotada ou não empacotada
- Microprogramação por hardware ou por software
- Codificação direta ou indireta

Essas classificações referem-se ao formato das microinstruções. Nenhum desses termos tem sido usado de forma precisa e consistente na literatura. Entretanto, examinar esses pares de características serve para dar uma idéia das alternativas de projeto de microinstruções. Nos parágrafos a seguir, examinamos primeiro uma questão básica de projeto subjacente a qualquer desses pares de características e, então, descrevemos os conceitos sugeridos por cada par.

Na proposta original de Wilkes (1951), cada bit de uma microinstrução produzia diretamente um sinal de controle ou um bit do endereço da próxima microinstrução. Vimos na seção precedente que, com esquemas mais complexos de seqüenciamento de endereços, é possível usar menor número de bits na microinstrução. Esses esquemas requerem um módulo de lógica de seqüenciamento mais complexo. Um compromisso similar ocorre em relação à porção da microinstrução referente a sinais de controle. Codificando a informação de controle e decodificando essa informação posteriormente para produzir os sinais de controle correspondentes, é possível economizar bits na palavra de controle.

Como essa codificação pode ser feita? Para responder a essa questão, suponha que o número de sinais de controle distintos a serem gerados pela unidade de controle, internos e externos, seja igual a K. No esquema de Wilkes, K bits da microinstrução seriam dedicados para esse fim. Isso permite gerar 2^K combinações possíveis de sinais de controle durante qualquer ciclo de microinstrução. O número de bits de sinal pode ser reduzido se observarmos que nem todas as possíveis combinações serão usadas. Alguns exemplos são:

- Duas fontes não podem ser dirigidas para o mesmo destino (por exemplo, C_2 e C_8 na Figura 14.5).
- Um registrador não pode ser simultaneamente fonte e destino (por exemplo, C_5 e C_{12} na Figura 14.5)
- Apenas um padrão de bits de controle pode ser apresentado à ULA em cada instante.
- Somente um padrão de sinais de controle pode ser apresentado ao barramento de controle externo em cada instante.

Portanto, devemos listar todas as combinações permitidas de sinais para um dado processador, obtendo um número de possibilidades $Q < 2^K$. Essas combinações podem ser codificadas usando $\log_2 Q$ bits, onde $(\log_2 Q) < K$. Isso seria a codificação mais compacta possível que preserva todas as combinações de sinais de controle permitidas. Essa forma de codificação não é usada na prática por duas razões:

- Ela é difícil de programar como um esquema de decodificação puro (de Wilkes). Esse ponto é discutido mais adiante.
- Ela requer um módulo de controle complexo e, portanto, lento.

Em vez disso, dois tipos de compromisso são adotados:

- É usado um número de bits maior que o estritamente necessário para expressar as combinações de sinais permitidas.
- Algumas combinações que são fisicamente possíveis não podem ser codificadas.

Esse último tipo de compromisso tem o efeito de reduzir o número de bits. Como resultado, o número de bits usado é maior que $\log_2 Q$ bits.

Na próxima subseção, são discutidas técnicas de codificação específicas. O restante desta subseção trata de efeitos da codificação e de vários termos usados para descrevê-la.

Com base na descrição precedente, podemos ver que a escolha do formato da parte da microinstrução correspondente a sinais de controle pode variar dentro de um determinado espectro de possibilidades: em um extremo, existe um bit para cada sinal de controle; no outro extremo, é usada uma codificação altamente compacta. A Tabela 15.4 mostra outras características de uma unidade de controle microprogramada que variam dentro de um determinado espectro, sendo esse espectro determinado, em última instância, pelo grau de codificação.

O segundo par de itens da tabela é bastante óbvio. O esquema de Wilkes é o que requer maior número de bits. Deve ser também aparente que esse extremo apresenta a visão mais detalhada do hardware: todo sinal de controle é controlável individualmente pelo microprogramador. A codificação é feita para agregar funções ou recursos, dando ao programador uma visão do processador de um nível mais alto e menos detalhado. Além disso, a codificação é projetada para facilitar a atividade de microprogramação, uma vez que a tarefa de entender e combinar o uso de todos os sinais de controle não é nada fácil. Como foi mencionado anteriormente, uma das conseqüências da codificação é, tipicamente, evitar o uso de certas combinações permitidas.

O parágrafo anterior discute o projeto de microinstruções sob o ponto de vista do microprogramador. Mas o grau de codificação pode também ser analisado com relação a seu efeito sobre o hardware. Com um formato não codificado, pouca ou nenhuma lógica de decodificação é requerida; cada bit gera um sinal de controle particular. Quanto mais compacto e agressivo é o esquema de codificação usado, mais complexa é a lógica de decodificação necessária. Isso pode, por sua vez, afetar o desempenho: mais tempo é requerido para propagar sinais por meio das portas de um módulo de lógica de controle mais complexo. Portanto, a execução de microinstruções codificadas consome mais tempo que a de microinstruções não codificadas.

Tabela 15.4 Espectro de microinstruções

Características	
Não codificada	Altamente codificada
Muitos bits	Poucos bits
Visão detalhada do hardware	Visão agregada do hardware
Difícil de programar	Fácil de programar
Concorrência explorada completamente	Concorrência não explorada completamente
Pouca ou nenhuma lógica de controle	Lógica de controle complexa
Execução rápida	Execução lenta
Otimiza o desempenho	Otimiza a programação
Terminologia	
Empacotada	Não empacotada
Horizontal	Vertical
Hard	Soft

Portanto, todas as características relacionadas na Tabela 15.4 abrangem um espectro de estratégias de projeto. Em geral, um projeto que se aproxima da extrema esquerda do espectro tem como objetivo otimizar o desempenho da unidade de controle. Projetos orientados para a extrema direita estão mais voltados para a otimização do processo de microprogramação. De fato, conjuntos de microinstruções próximos da extremidade direita do espectro são semelhantes a conjuntos de instruções de máquina. Um bom exemplo é o projeto do LSI-11, descrito mais tarde nesta seção. Tipicamente, quando o objetivo é simplesmente implementar uma unidade de controle, o projeto tende para a extremidade esquerda do espectro. O projeto do IBM 3033 pertence a essa categoria. Como discutiremos mais tarde, alguns sistemas permitem que usuários construam diferentes microprogramas usando o mesmo mecanismo de microinstrução. Nesses casos, o projeto tende para a extremidade direita do espectro.

Podemos agora lidar com a terminologia introduzida anteriormente. A Tabela 15.4 indica como os três pares de características mencionados se relacionam com o espectro de microinstruções. Essencialmente, todos esses pares descrevem a mesma coisa, mas enfatizam diferentes características de projeto.

O grau de empacotamento está relacionado com o grau de identificação entre uma tarefa de controle e os bits específicos da microinstrução. À medida que os bits são mais *empacotados*, um menor número de bits contém mais informação. Portanto, empacotamento significa codificação. Os termos *horizontal* e *vertical* estão relacionados ao tamanho relativo de microinstruções. Siewiorek, Bell e Newell (1982) sugerem, como regra geral, que microinstruções verticais têm tamanho entre 16 e 40 bits e microinstruções horizontais têm tamanho entre 40 e 100 bits. Os termos microprogramação *hard* e *soft* são usados para sugerir o grau de proximidade dos sinais de controle e do leiaute de hardware subjacentes. Microprogramas *hard* geralmente são fixos e armazenados em memória apenas de leitura. Microprogramas *soft* podem ser alterados mais facilmente e são mais atraentes para o microprogramador.

O outro par de termos mencionado no início desta subseção refere-se a codificação direta *versus* codificação indireta, que são tratadas a seguir.

Codificação de microinstrução

Na prática, unidades de controle microprogramadas não são projetadas usando um formato de microinstrução horizontal ou vertical puro. Pelo menos algum grau de codificação é usado para reduzir a memória de controle e simplificar a tarefa de microprogramação.

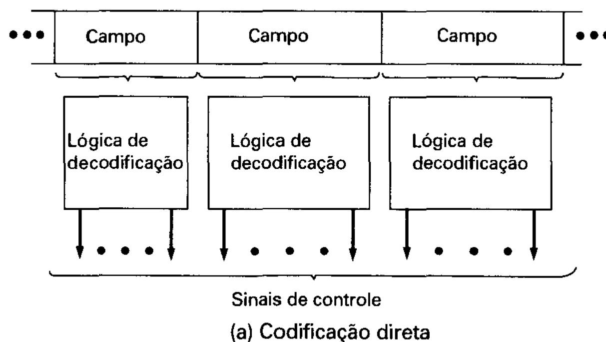
A técnica básica de codificação é ilustrada na Figura 15.11a. A microinstrução é organizada como um conjunto de campos. Cada campo contém um código que, depois de decodificado, ativa um ou mais sinais de controle.

Consideremos as implicações desse esquema. Quando a microinstrução é executada, cada campo é decodificado e gera sinais de controle. Portanto, N campos podem especificar N ações simultâneas. Cada ação resulta na ativação de um ou mais sinais de controle. Geralmente, mas nem sempre, gostaríamos de projetar um formato de microinstrução de modo que cada sinal de controle seja ativado por apenas um campo. É claro que cada sinal de controle seja ativado por pelo menos um campo.

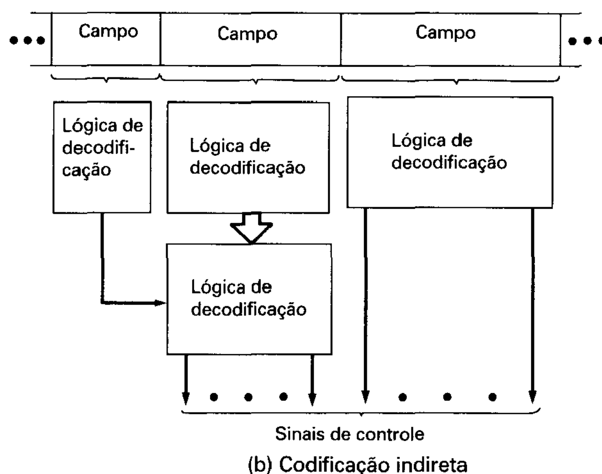
Considere agora um campo individual. Um campo consistindo de L bits pode conter um dentre 2^L códigos, cada um dos quais pode codificar um padrão de sinais de controle distinto. Como apenas um código pode aparecer em um campo de cada vez, esses códigos são mutuamente exclusivos e as ações causadas por eles são mutuamente exclusivas.

O projeto de um formato de microinstrução codificada pode ser formulado nos seguintes termos:

- Organize o formato em campos independentes. Isto é, cada campo especifica um conjunto de ações (padrão de sinais de controle), de modo que ações definidas por diferentes campos podem ocorrer simultaneamente.
- Defina cada campo de maneira que ações alternativas que podem ser especificadas pelo campo sejam mutuamente exclusivas. Isto é, apenas uma das ações especificadas por um dado campo ocorrem em um dado instante.



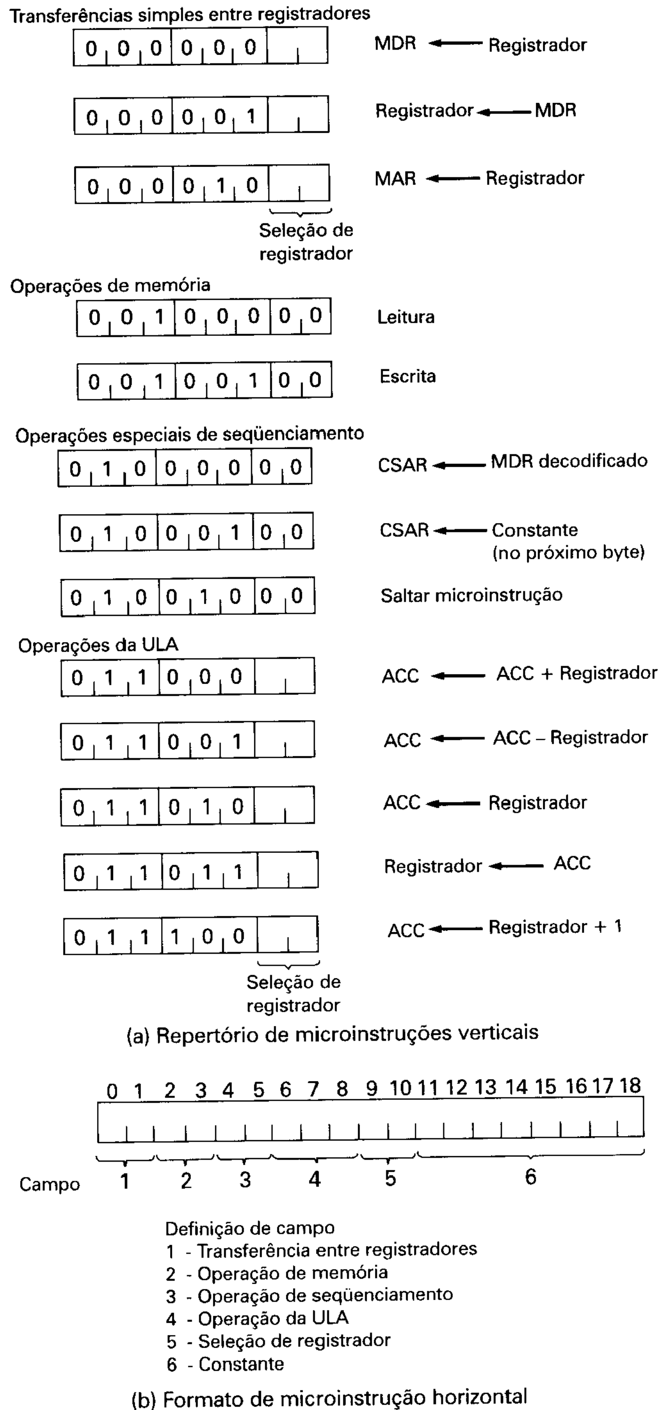
(a) Codificação direta



(b) Codificação indireta

Figura 15.11 Codificação de microinstrução.

Duas abordagens podem ser adotadas para organizar as microinstruções codificadas em campos: codificação funcional e codificação por recursos. O método de *codificação funcional* identifica as funções da máquina e designa campos para cada tipo de função. Por exemplo, se várias fontes podem ser usadas para transferir dados para o acumulador, um campo pode ser destinado a esse propósito, com cada código especificando uma fonte diferente. A *codificação por recursos* vê a máquina como consistindo de um conjunto de recursos independentes e designa um campo para cada um (por exemplo, E/S, memória, ULA).



(a) Repertório de microinstruções verticais

(b) Formato de microinstrução horizontal

Figura 15.12 Formatos alternativos de microinstrução para uma máquina simples.

Outro aspecto da codificação é se ela é direta ou indireta (Figura 15.11b). Com a codificação indireta, um campo é usado para determinar a interpretação de outro campo. Por exemplo, considere uma ULA capaz de efetuar oito operações aritméticas diferentes e oito operações de deslocamento diferentes. Um campo de 1 bit pode ser usado para indicar uma operação aritmética ou de deslocamento; um campo de 3 bits pode ser usado para indicar a operação. Essa técnica geralmente envolve dois níveis de decodificação, aumentando o atraso de propagação de sinais.

A Figura 15.12 é um exemplo simples desses conceitos. Suponha um processador com um único acumulador e vários registradores internos, tais como um contador de programa e um registrador temporário para entrada da ULA. A Figura 15.12a mostra um formato de microinstrução altamente vertical. Os 3 primeiros bits indicam o tipo de operação, os 3 próximos bits codificam a operação e os 2 últimos bits selecionam um registrador interno. A Figura 15.12b adota uma abordagem mais horizontal, embora a codificação ainda seja usada. Nesse caso, as diferentes funções aparecem em diferentes campos.

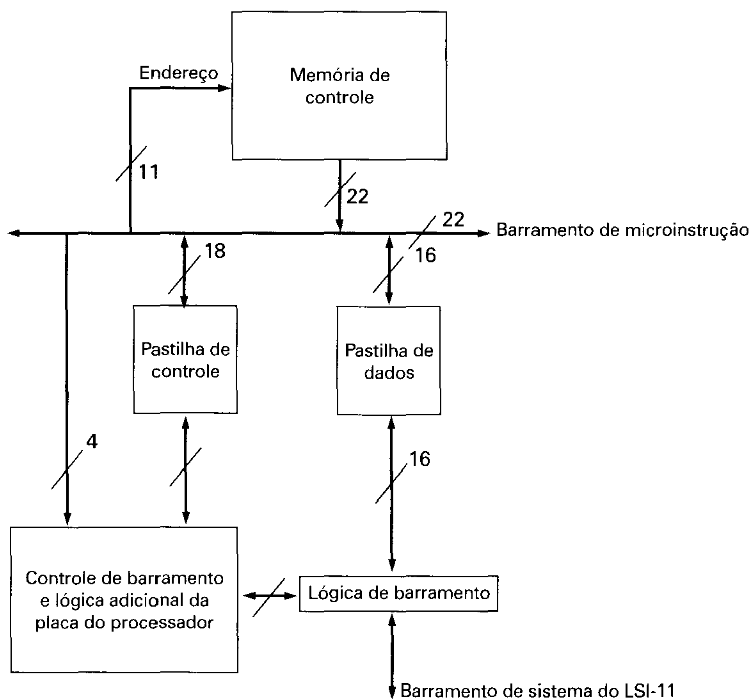


Figura 15.13 Diagrama de bloco simplificado do processador LSI-11.

Execução de microinstrução no LSI-11

O LSI-11 (Sebern, 1976) é um bom exemplo de uso da abordagem de microinstrução vertical. Vamos primeiro examinar a organização da unidade de controle e, então, o formato de microinstrução.

Organização da unidade de controle do LSI-11

O LSI-11 é o primeiro membro da família PDP-11 com processador constituído de uma única placa de circuito integrado. Essa placa contém três pastilhas LSI, um barramento interno, conhecido como *barramento de microinstrução* (MIB), e lógica adicional de interface.

A Figura 15.13 mostra, de forma simplificada, a organização do processador LSI-11. As três pastilhas que compõem o processador são a pastilha de controle, a pastilha de dados e a pastilha de memória de controle. A pastilha de dados contém uma ULA de 8 bits, 26 registradores de 8 bits e área para armazenamento de vários códigos de condição. Dezesseis dos registradores são usados para implementar os 8 registradores de propósito geral de 16 bits do PDP-11. Outros registradores incluem a palavra de estado de programa, o registrador de endereço de memória (MAR) e o registrador buffer de memória (MBR). Como a ULA lida apenas com 8 bits de cada vez, são requeridos dois passos por meio da ULA para implementar operações aritméticas de 16 bits do PDP-11. Isso é controlado pelo microprograma.

A pastilha (ou pastilhas) de memória de controle tem uma largura de 22 bits. A pastilha de controle contém a lógica para o seqüenciamento e a execução de microinstruções. Ela contém o registrador de endereço de controle, o registrador de dados de controle e uma cópia do registrador de instrução da máquina.

O barramento de microinstrução (MIB) conecta todos esses componentes. Durante a busca de microinstrução, a pastilha de controle gera um endereço de 11 bits no barramento de microinstrução. A memória de controle é lida, produzindo uma microinstrução de 22 bits, que é colocada no MIB. Os 16 bits de ordem mais baixa vão para a pastilha de dados; os 18 bits de ordem mais baixa vão para a pastilha de controle. Os 4 bits de ordem mais alta controlam funções especiais da placa do processador.

A Figura 15.14 fornece uma visão mais detalhada, embora ainda simplificada, da unidade de controle do LSI-11: a figura ignora os limites de pastilhas individuais. O esquema de seqüenciamento de endereços descrito na Seção 15.2 é implementado em dois módulos. O controle global é feito pelo módulo de seqüenciamento de microprograma, que é capaz de incrementar o registrador de endereço de microinstrução e efetuar desvios incondicionais. As outras formas de cálculo de endereço são realizadas por um vetor de tradução separado. Esse vetor de tradução é um circuito combinatório que gera um endereço baseado na microinstrução, na instrução de máquina, no contador de microinstrução e no registrador de interrupção.

O vetor de tradução atua nos seguintes casos:

- Quando o código de operação é usado para determinar o início de uma microrrotina.
- Em instantes apropriados, quando os bits de modo de endereçamento da microinstrução são testados para efetuar o endereçamento apropriado.
- Periodicamente, para testar condições de interrupção.
- Quando são avaliadas microinstruções de desvio condicional.

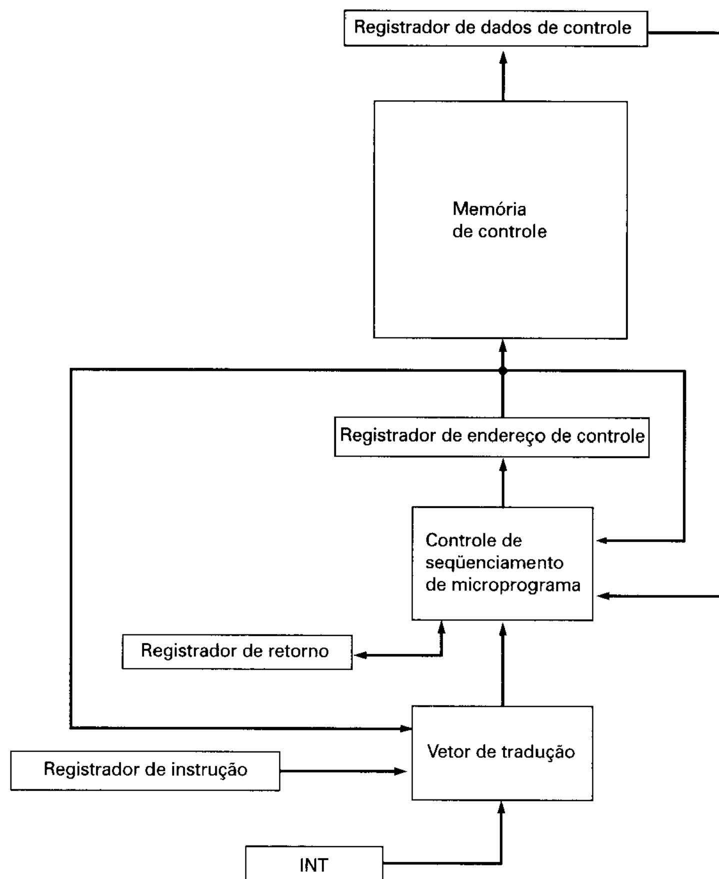


Figura 15.14 Organização da unidade de controle do LSI-11.

Formato de microinstrução do LSI-11

O LSI-11 usa um formato de microinstrução extremamente vertical, com tamanho de 22 bits. O conjunto de microinstruções é muito semelhante ao conjunto de instruções de máquina do PDP-11 que ele implementa. Esse projeto tinha como objetivo otimizar o desempenho da unidade de controle, com a restrição de ser um projeto de microinstruções verticais, que facilitasse a microprogramação. A Tabela 15.5 relaciona algumas das microinstruções do LSI-11.

A Figura 15.5 mostra o formato das microinstruções do LSI-11 de 22 bits. Os 4 bits de mais alta ordem controlam funções especiais da placa do processador. O bit de tradução habilita o vetor de tradução para verificar a existência de interrupções pendentes. O bit de carga de registrador de retorno é usado ao final de uma microrrotina, para fazer com que o endereço da próxima microinstrução seja carregado a partir do registrador de retorno.

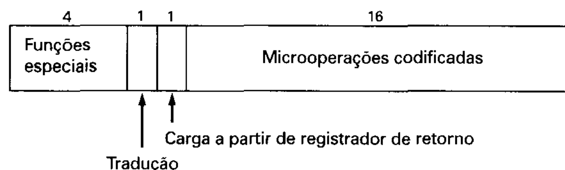
Tabela 15.5 Algumas microinstruções do LSI-11

Operações aritméticas	**Operações gerais**
Soma palavra (byte, literal)	Move palavra (byte)
Testa palavra (byte, literal)	Desvio incondicional
Incrementa palavra (byte) de 1	Retorno
Incrementa palavra (byte) de 2	Desvio condicional
Troca sinal de palavra (byte)	Atribui valor 1 (0) a bits de condição
Incrementa (decrementa) byte condicionalmente	Carrega byte menos significativo de G Move palavra (byte) condicionalmente
Soma palavra (byte) condicionalmente	
Soma palavra (byte) com 'vai-um'	**Operações de E/S**
Soma dígitos condicionalmente	Entrada de palavra (byte)
Subtrai palavra (byte)	Saída de palavra (byte) de estado
Compara palavra (byte, literal)	Leitura
Subtrai palavra (byte) com 'vai-um'	Escrita
Decrementa palavra (byte) de 1	Leitura (escrita) e incremento de palavra (byte) de 1
Operações lógicas	Leitura (escrita) e incremento de palavra (byte) de 2
AND de palavra (byte)	Reconhecimento de leitura (escrita)
Testa palavra (byte)	Saída de palavra (byte, estado)
OR de palavra (byte)	
XOR de palavra (byte)	
Atribui zero a palavra (byte)	
Desloca palavra (byte) para direita (esquerda) com (sem) 'vai-um'	
Complementa palavra (byte)	

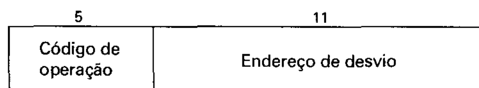
Os 16 bits restantes são usados para microoperações codificadas. O formato é semelhante ao de instruções de máquina, tendo um campo de código de operação de tamanho variável e um ou mais operandos.

Execução de microinstrução no IBM 3033

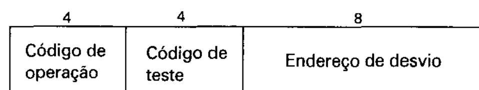
A memória de controle padrão do IBM 3033 consiste de 4 K palavras. A primeira metade da memória de controle (0000-07FF) contém microinstruções de 108 bits e a segunda me tade (0800-0FFF) é usada para armazenar microinstruções de 126 bits. O formato é mostrado na Figura 15.16. Embora o formato seja bastante horizontal, a codificação é ainda usada amplamente. Os principais campos do formato são listados na Tabela 15.6.



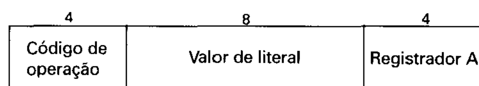
(a) Formato de microinstrução completa do LSI-11



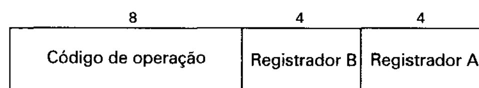
Formato de microinstrução de desvio incondicional



Formato de microinstrução de desvio condicional



Formato de microinstrução de literal



Formato de microinstrução de registrador

(b) Formato da parte codificada das microinstruções do LSI-11

Figura 15.15 Formato de microinstrução do LSI-11.

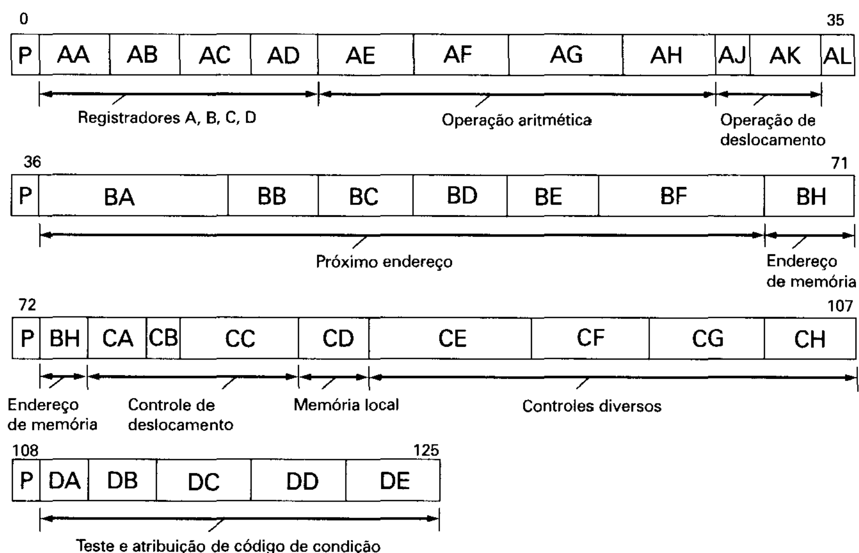


Figura 15.16 Formato de microinstrução do IBM 3033.

Tabela 15.6 Campos de controle de microinstrução do IBM 3033

	Campos de Controle da ULA
AA(3)	Carrega registrador A a partir de um dos registradores de dados
AB(3)	Carrega registrador B a partir de um dos registradores de dados
AC(3)	Carrega registrador C a partir de um dos registradores de dados
AD(3)	Carrega registrador D a partir de um dos registradores de dados
AE(4)	Direciona bits especificados do registrador A para a ULA
AF(4)	Direciona bits especificados do registrador B para a ULA
AG(5)	Especifica operação aritmética da ULA sobre a entrada A
AH(4)	Especifica operação aritmética da ULA sobre a entrada B
AJ(1)	Especifica o registrador D ou B como entrada para a ULA no lado B
AK(4)	Direciona saída de operação aritmética como entrada do deslocador
CB(1)	Ativa deslocador
CC(5)	Especifica funções lógicas e de 'vai-um'
CE(7)	Especifica quantidade de deslocamento
CA(3)	Carrega registrador F
	Campos de Seqüenciamento e de Desvio
AL(1)	Termina operação e efetua desvio
BA(8)	Ativa os bits de mais alta ordem (00–07) do registrador de endereço de controle
BB(4)	Especifica condição para ativar o bit 8 do registrador de endereço de controle
BC(4)	Especifica condição para ativar o bit 9 do registrador de endereço de controle
BD(4)	Especifica condição para ativar o bit 10 do registrador de endereço de controle
BE(4)	Especifica condição para ativar o bit 11 do registrador de endereço de controle
BF(4)	Especifica condição para ativar o bit 12 do registrador de endereço de controle

A ULA opera sobre entradas obtidas de quatro registradores dedicados, A, B, C e D, não visíveis para o usuário. O formato de microinstrução contém campos para carregar esses registradores a partir de registradores visíveis para o usuário, para efetuar uma função da ULA e para especificar um registrador visível para o usuário armazenar o resultado. Há também campos para transferir dados entre os registradores e a memória.

O mecanismo de seqüenciamento do IBM 3033 foi discutido brevemente na Seção 15.2.

15.4 TI 8800

O Texas Instruments 8800 Software Development Board (SDB) é uma placa de computador programável de 32 bits. O sistema tem memória de controle gravável, implementada em RAM, e não atinge o desempenho ou a densidade de sistemas microprogramados que usam memória ROM. Entretanto, ele é útil para o desenvolvimento de protótipos e para propósitos educacionais.

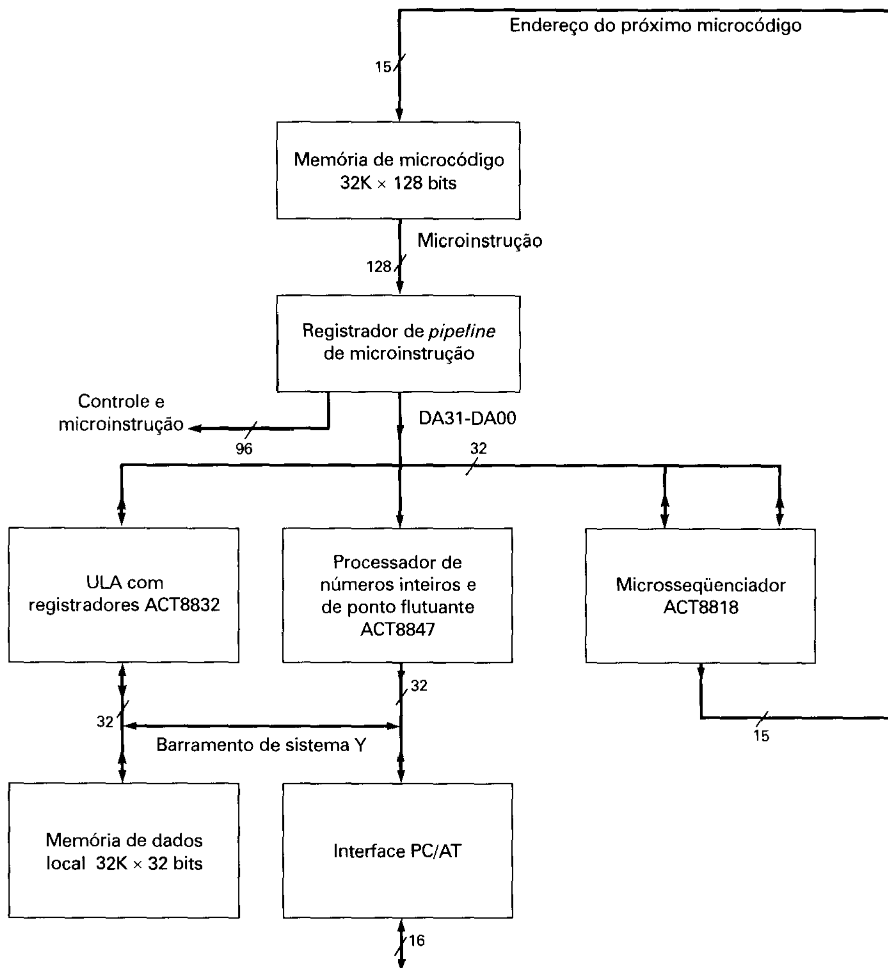


Figura 15.17 Diagrama de blocos do TI 8800.

O SDB 8800 consiste dos seguintes componentes (Figura 15.17):

- Memória de microcódigo
- Microsseqüenciador
- ULA de 32 bits
- Processador de números inteiros e de ponto flutuante
- Memória de dados local

Dois barramentos conectam os componentes internos do sistema. O barramento DA provê dados do campo de dados da microinstrução para a ULA, o processador de ponto flutuante e o microsseqüenciador. Neste último caso, o dado é um endereço a ser usado para uma instrução de desvio. O barramento também pode ser usado pela ULA ou pelo microsseqüenciador para prover dados para outros componentes. O barramento de sistema Y conecta a ULA e o processador de ponto flutuante à memória local e aos módulos externos, por meio da interface PC/AT.

A placa se encaixa em um computador hospedeiro compatível com IBM PC. O computador hospedeiro provê uma plataforma adequada para montagem e depuração do microcódigo.

Formato de microinstrução

O formato de microinstrução do 8800 consiste de 128 bits, divididos em 30 campos funcionais, como indicado na Tabela 15.7. Cada campo consiste de um ou mais bits, e os campos são agrupados em cinco categorias principais:

- Controle da placa
- Pastilha do processador de números inteiros e de ponto flutuante 8847
- ULA com registradores 8832
- Microsseqüenciador 8818
- Campo de dados do WCS

Como indicado na Figura 15.17, os 32 bits do campo de dados WCS são enviados para o barramento DA, para servirem como dado para a ULA, para o processador de ponto flutuante ou para o microsseqüenciador. Os outros 96 bits (campos 1 a 27) da microinstrução são sinais de controle enviados diretamente para o módulo apropriado. Para simplificar, essas outras conexões não são mostradas na Figura 15.17.

Os seis primeiros campos de operação lidam com operações de controle da placa, e não com o controle de componentes individuais. As operações de controle incluem:

- Seleção de códigos de condição para controle de seqüenciamento. O primeiro bit do campo 1 indica se um bit de condição deve ser ativado ou desativado, e os 4 bits restantes indicam o bit de condição a ser atualizado.
- Envio de requisição de E/S para a interface PC/AT.
- Habilitação de operações de leitura/escrita na memória local.
- Determinação da unidade que detém o controle do barramento de sistema Y. Um dos quatro dispositivos conectados ao barramento é selecionado (Figura 15.17).

Os últimos 32 bits constituem o campo de dados, que contém informação específica para uma instrução particular.

Tabela 15.7 Formato de microinstrução do TI 8800

Número do campo	Número de bits	Descrição
		Controle da placa
1	5	Seleciona entrada de código de condição
2	1	Habilita/desabilita sinal externo de requisição de E/S
3	2	Habilita/desabilita operações de leitura/escrita na memória local
4	1	Carrega/não carrega registrador de estado
5	2	Determina unidade que detém controle do barramento de sistema Y
6	2	Determina unidade que detém controle do barramento DA

Tabela 15.7 Formato de microinstrução do TI 8800 (*continuação*)

Número do campo	Número de bits	Descrição
		Pastilha do processador de números inteiros e de ponto flutuante 8847
7	1	Controle do registrador C: pulsa relógio, não pulsa relógio
8	1	Seleciona bits mais significativos ou bits menos significativos do barramento Y
9	1	Fonte de dados do registrador C: ULA, multiplexador
10	4	Seleciona modo IEEE ou FAST para a ULA e MUL
11	8	Seleciona fonte de operandos de dado: registradores RA, registradores RB, registrador P, registrador S, registrador C
12	1	Controle do registrador RB: envia sinal de relógio, não envia sinal de relógio
13	1	Controle do registrador RA: envia sinal de relógio, não envia sinal de relógio
14	2	Confirmação de fonte de dado
15	2	Habilita/desabilita registradores da *pipeline*
16	11	Função da ULA do 8847
		ULA com registradores 8832
17	2	Habilita/desabilita escrita de dado de saída para registrador selecionado: metade mais significativa, metade menos significativa
18	2	Seleciona fonte de dados para o banco de registradores: barramento DA, barramento DB, saída do multiplexador da ULA, barramento do sistema
19	3	Modificador de instrução de deslocamento
20	1	Força/não força 'vai-um' para a ULA
21	2	Determina configuração da ULA: 32, 16 ou 8 bits
22	2	Seleciona entrada para o multiplexador S: banco de registradores, barramento, registrador MQ
23	1	Seleciona entrada para o multiplexador R: banco de registradores, barramento DA
24	6	Seleciona registrador do banco C para escrita
25	6	Seleciona registrador do banco B para escrita
26	6	Seleciona registrador do banco A para escrita
27	8	Função da ULA
		Microsseqüenciador 8818
28	12	Sinais de entrada de controle do 8818
		Campo de dados do WCS
29	16	Bits mais significativos do campo de dados de memória de controle gravável
30	16	Bits menos significativos do campo de dados de memória de controle gravável

Os demais campos da microinstrução são mais bem descritos no contexto do dispositivo que eles controlam. No restante desta seção, discutimos o microsseqüenciador e a ULA com registradores. A unidade de ponto flutuante não é abordada, uma vez que não introduz nenhum conceito novo.

Microsseqüenciador

A principal função do microsseqüenciador 8818 é gerar o endereço da próxima microinstrução do microprograma a ser executada. Esse endereço de 15 bits é fornecido para a memória de microcódigo (Figura 15.17).

O próximo endereço pode ser selecionado de uma dentre cinco fontes:

1. O registrador contador de microinstruções (MPC), usado para instruções de repetição (uso repetido do mesmo endereço) e instruções de continuação (o endereço é incrementado de 1).
2. A pilha, que provê suporte a chamadas de sub-rotinas de microprograma, assim como a laços de repetição e retorno de interrupções.
3. As portas DRA e DRB, que oferecem dois caminhos adicionais vindos do hardware externo, através dos quais podem ser gerados endereços de microprograma. Essas duas portas são conectadas aos 16 bits mais significativos e aos 16 bits menos significativos do barramento DA, respectivamente. Isso possibilita ao microsseqüenciador obter o próximo endereço de microinstrução do campo de dados do WCS da microinstrução corrente ou de um resultado calculado pela ULA.
4. Registradores contadores RCA e RCB, que podem ser usados como área adicional de armazenamento de endereço.
5. Uma entrada externa na porta bidirecional Y, para prover suporte a interrupções externas.

A Figura 15.18 é um diagrama de blocos lógico do 8818. O dispositivo consiste dos seguintes grupos funcionais principais:

- Um contador de microinstruções (MPC) de 16 bits, consistindo de um registrador e de lógica para incrementar esse registrador.
- Dois registradores contadores, RCA e RCB, para contar iterações e laços de repetição, armazenar endereços de desvio ou direcionar dispositivos externos.
- Uma pilha de 65 palavras de 16 bits, que provê suporte para chamada de sub-rotinas de microprograma e para interrupções.
- Um registrador de retorno de interrupção e a saída Y possibilitam o processamento de interrupção no nível de microinstrução.
- Um multiplexador da saída Y, por meio do qual o próximo endereço pode ser selecionado dos registradores MPC, RCA, RCB, dos barramentos externos DRA e DRB e da pilha.

Registradores/contadores

Os registradores RCA e RCB podem ser carregados por meio do barramento DA, seja a partir da microinstrução corrente seja da saída da ULA. O valor contido nesses registradores pode ser usado como um contador para controlar o fluxo de execução e pode ser decrementado automaticamente quando é usado. O valor pode também ser usado como endereço de microinstrução, para ser fornecido ao mutiplexador da saída Y. Os dois registradores podem ser controlados de forma independente durante um mesmo ciclo de microinstrução, com exceção do decremento simultâneo dos dois registradores.

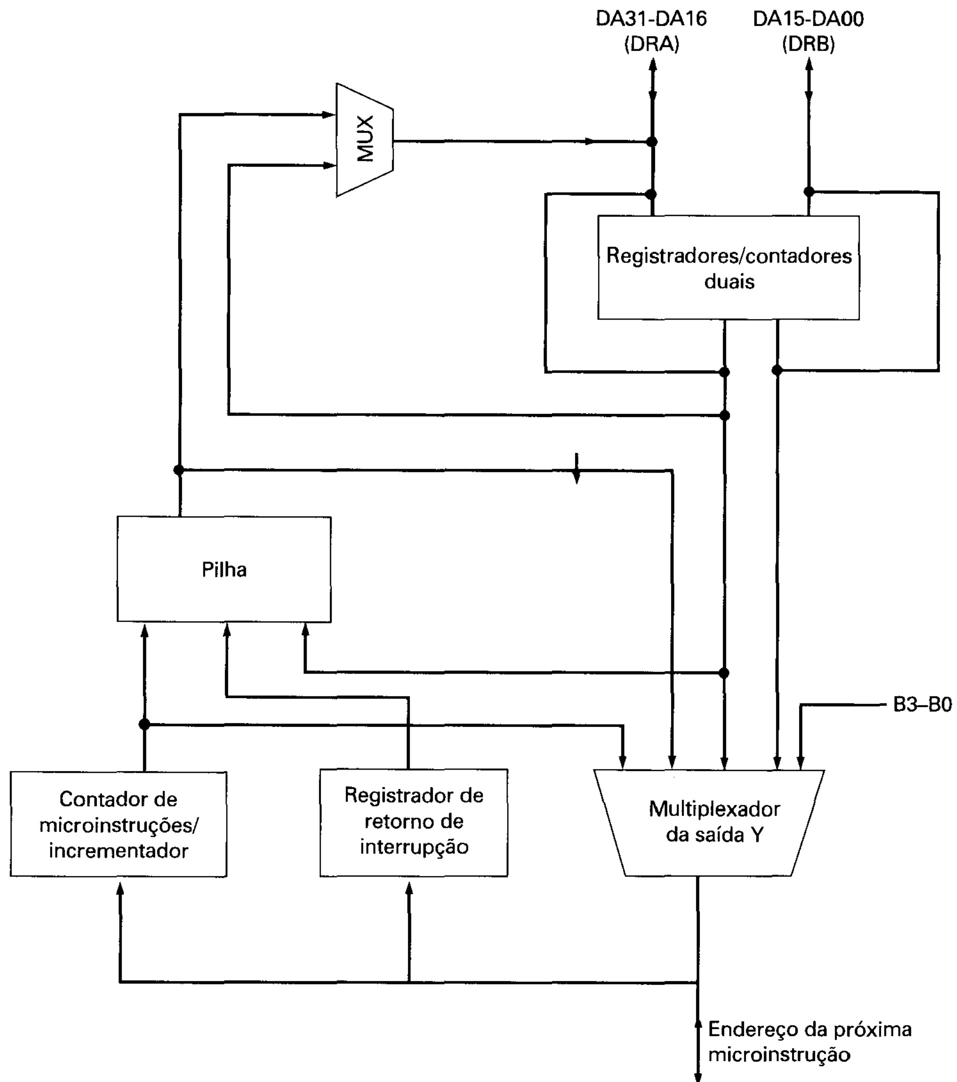


Figura 15.18 Microsseqüenciador TI 8818.

Pilha

A pilha possibilita múltiplos níveis de chamadas ou interrupções aninhadas e pode ser usada para prover suporte para desvios e laços de repetição. Lembre-se de que essas operações se referem à unidade de controle, e não ao processador como um todo, e que os endereços envolvidos são endereços de microinstruções residentes na memória de controle.

Existem seis possíveis operações de pilha:

1. Esvazia pilha, que atualiza o apontador de topo da pilha com valor zero.
2. Desempilha, que decrementa o apontador de topo da pilha.

3. Empilha, que coloca no topo da pilha o conteúdo do MPC, do registrador de retorno de interrupção ou do barramento DRA, e incrementa o apontador de topo da pilha.
4. Leitura, que torna disponível no multiplexador da saída Y o endereço indicado pelo apontador de leitura.
5. Mantém, que faz com que o endereço contido no apontador de topo da pilha seja mantido inalterado.
6. Carrega apontador de topo da pilha, que carrega os sete bits menos significativos do barramento DRA no apontador de topo da pilha.

Controle do microsseqüenciador

O microsseqüenciador é controlado primariamente pelo campo 28 da instrução corrente, que tem 12 bits (Tabela 15.7). Esse campo consiste dos seguintes subcampos:

- **OSEL (1 bit):** seleção de saída. Determina que valor será colocado na saída do multiplexador que alimenta o barramento DRA (canto superior esquerdo da Figura 15.18). A saída é selecionada como vinda da pilha ou do registrador RCA. O DRA então serve como entrada para o multiplexador da saída Y ou para o registrador RCA.
- **SELDR (1 bit):** seleciona barramento DR. Se esse bit tem valor 1, é selecionado o barramento externo DA como entrada para os barramentos DRA/DRB. Se tem valor 0, é selecionada a saída do multiplexador do DRA para o barramento DRA (controlado pelo bit OSEL) e o conteúdo do registrador RCB para o barramento DRB.
- **ZEROIN (1 bit):** usado para indicar desvio condicional. O comportamento do microsseqüenciador dependerá, então, do código de condição selecionado no campo 1 (Tabela 15.7).
- **RC2-RC0 (3 bits):** controle de registrador. Esses bits determinam a mudança no conteúdo dos registradores RCA e RCB. Cada registrador pode permanecer com o mesmo valor, ser decrementado ou ser carregado a partir dos barramentos DRA/DRB.
- **S2-S0 (3 bits):** controle de pilha. Esses bits determinam a operação de pilha que deve ser efetuada.
- **MUX2-MUX0 (3 bits):** controle de saída. Esses bits, juntamente com o código de condição (caso usado), controlam o multiplexador da saída Y e, portanto, o próximo endereço de microinstrução. O multiplexador pode selecionar a saída a partir da pilha, dos barramentos DRA e DRB ou do registrador MPC.

O valor de cada um desses bits pode ser especificado separadamente pelo programador. Entretanto, tipicamente, isso não é feito. Em vez disso, o programador usa mnemônicos que denotam padrões de bits normalmente requeridos. A Tabela 15.8 lista os 15 mnemônicos usados para o campo 28. Um montador de microcódigo converte cada mnemônico no padrão de bits apropriado.

Como um exemplo, a instrução INC88181 é usada para selecionar a próxima instrução na seqüência, caso o código de condição correntemente selecionado seja igual a 1. Pela Tabela 15.8, temos

```
INC88181 = 000000111110
```

que decodifica diretamente em:

- **OSEL = 0:** seleciona RCA como a saída do multiplexador do barramento DRA; nesse caso, a seleção é irrelevante.
- **SELDR = 0:** como definido anteriormente; novamente, isso é irrelevante para essa instrução.
- **ZEROIN = 0:** combinado como valor do MUX, indica que o desvio não deve ser tomado.
- **R = 000:** mantém o valor corrente de RA e RB.
- **S = 111:** mantém o estado corrente da pilha.
- **MUX = 110:** escolhe o registrador MPC, se código de condição = 1, ou o DRA, se código de condição = 0.

Tabela 15.8 Bits de microinstrução do microsseqüenciador TI 8818 (campo 28)

Mnemônico	Valor	Descrição
RST8818	000000000110	Instrução de reinicialização
BRA88181	011000111000	Desvio para instrução do DRA
BRA88180	010000111110	Desvio para instrução do DRA
INC88181	000000111110	Instrução de continuação
INC88180	001000001000	Instrução de continuação
CAL88181	010000110000	Desvio para sub-rotina cujo endereço é especificado pelo DRA
CAL88180	010000101110	Desvio para sub-rotina cujo endereço é especificado pelo DRA
RET8818	000000011010	Retorno de sub-rotina
PUSH8818	000000110111	Empilha endereço de retorno de interrupção
POP8818	100000010000	Retorno de interrupção
LOADDRA	000010111110	Carrega contador DRA a partir do barramento DA
LOADDRB	000110111110	Carrega contador DRB a partir do barramento DB
LOADDRAB	000110111100	Carrega DRA/DRB
DECRDRA	010001111100	Decrementa contador DRA e desvia se diferente de zero
DECRDRB	010101111100	Decrementa contador DRB e desvia se diferente de zero

ULA com registradores

O 8832 é uma ULA de 32 bits com 64 registradores que pode ser configurada para operar como quatro ULAs de 8 bits, duas ULAs de 16 bits ou uma ULA de 32 bits.

O 8832 é controlado por 39 bits, que compõem os campos 17 a 27 da microinstrução (Tabela 15.7); esses bits são fornecidos à ULA como sinais de controle. Além disso, como indicado na Figura 15.17, o 8832 tem conexões externas com o barramento DA de 32 bits e com o barramento de sistema Y, também de 32 bits. Entradas originárias do barramento DA podem ser submetidas simultaneamente como entrada para um banco de registradores de 64 palavras e para o módulo de lógica da ULA. Os resultados das operações da ULA e de operações de deslocamento são direcionados para o barramento DA ou para o barramento de sistema Y. Os resultados podem também ser direcionados de volta para o banco de registradores interno.

As portas de endereços de 6 bits possibilitam buscar dois operandos e escrever um valor no banco de registradores, simultaneamente. O registrador MQ e o deslocador MQ podem ser configurados para funcionar independentemente, para implementar operações de deslocamento sobre valores de 8 bits, 16 bits ou 32 bits.

Os campos 17 a 26 de cada microinstrução controlam o fluxo de dados dentro do 8832 e entre o 8832 e o ambiente externo. Esses campos são:

17. *Habilitação de escrita.* Os dois bits deste campo especificam uma escrita de 32 bits, ou dos 16 bits mais significativos, ou dos 16 bits menos significativos ou nenhuma escrita no banco de registradores. O registrador destino é definido pelo campo 24.

18. *Seleção fonte de escrita sobre banco de registradores.* Se deve ser efetuada uma escrita sobre um banco de registradores, esses dois bits especificam a fonte: barramento DA, barramento DB, saída da ULA ou barramento de sistema Y.

19. *Modificador de instrução de deslocamento.* Especifica opções relativas ao fornecimento de bits de preenchimento e à leitura de bits que são deslocados em uma operação de deslocamento.

20. *Vai-um*: Esse bit indica se é somado 1 ao bit menos significativo do resultado obtido nessa operação da ULA.

21. *Modo de configuração da ULA.* O 8832 pode ser configurado para operar como uma única ULA de 32 bits, como duas ULAs de 16 bits ou como quatro ULAs de 8 bits.

22. *Entrada S.* As entradas do módulo de lógica da ULA são fornecidas por dois multiplexadores internos, chamados de multiplexadores S e R. Esse campo seleciona a entrada como sendo fornecida pelo multiplexador S: banco de registradores, barramento DB ou registrador MQ. O registrador fonte é definido pelo campo 25.

23. *Entrada R.* Seleciona a entrada da ULA como sendo provida pelo multiplexador R: banco de registradores ou barramento DA.

24. *Registrador destino.* Endereço do registrador no banco de registradores a ser usado como operando destino.

25. *Registrador fonte.* Endereço do registrador no banco de registradores a ser usado como operando fonte, fornecido pelo multiplexador S.

26. *Registrador fonte.* Endereço do registrador no banco de registradores a ser usado como operando fonte, fornecido pelo multiplexador R.

Finalmente, o campo 27 é um código de operação de 8 bits, que especifica a função lógica ou aritmética a ser executada pela ULA. A Tabela 15.9 lista as diferentes operações que podem ser executadas.

Tabela 15.9 Campo de instrução da ULA com registradores TI 8832 (campo 27)

Grupo 1		Função
ADD	H#01	R + S + Cn
SUBR	H#02	(NOT R) + S + Cn
SUBS	H#03	R = (NOT S) + Cn
INSC	H#04	S + Cn
INCNS	H#05	(NOT S) + Cn
INCR	H#06	R + Cn
INCNR	H#07	(NOT R) + Cn

Tabela 15.9 Campo de instrução da ULA com registradores TI 8832 (campo 27) (*continuação*)

Grupo 1		**Função**
XOR	H#09	R XOR S
AND	H#0A	R AND S
OR	H#0B	R OR S
NAND	H#0C	R NAND S
NOR	H#0D	R NOR S
ANDNR	H#0E	(NOT R) AND S
Grupo 2		**Função**
SRA	H#00	Deslocamento aritmético de precisão simples para a direita
SRAD	H#10	Deslocamento aritmético de precisão dupla para a direita
SRL	H#20	Deslocamento lógico de precisão simples para a direita
SRLD	H#30	Deslocamento lógico de precisão dupla para a direita
SLA	H#40	Deslocamento aritmético de precisão simples para a esquerda
SLAD	H#50	Deslocamento aritmético de precisão dupla para a esquerda
SLC	H#60	Deslocamento circular de precisão simples para a esquerda
SLCD	H#70	Deslocamento circular de precisão dupla para a esquerda
SRC	H#80	Deslocamento circular de precisão simples para a direita
SRCD	H#90	Deslocamento circular de precisão dupla para a direita
MQSRA	H#A0	Deslocamento aritmético do registrador MQ para a direita
MQSRL	H#B0	Deslocamento lógico do registrador MQ para a direita
MQSLL	H#C0	Deslocamento lógico do registrador MQ para a esquerda
MQSLC	H#D0	Deslocamento circular do registrador MQ para a esquerda
LOADMQ	H#E0	Carrega registrador MQ
PASS	H#F0	Passa saída da ULA para barramento Y (sem operação de deslocamento)
Grupo 3		**Função**
SET1	H#08	Ativa bit 1
SET0	H#18	Ativa bit 0
TB1	H#28	Testa bit 1
TB0	H#38	Testa bit 0
ABS	H#48	Valor absoluto
SMTC	H#58	Sinal-magnitude ou Complemento de dois
ADDI	H#68	Soma imediato
SUBI	H#78	Subtrai imediato
BADD	H#88	Soma byte de R em S

Tabela 15.9 Campo de instrução da ULA com registradores TI 8832 (campo 27) (*continuação*)

Grupo 3		Função
BSUBS	H#98	Subtrai byte S de R
BSUBR	H#A8	Subtrai byte R de S
BINCS	H#B8	Incremento de byte de S
BINCNS	H#C8	Incremento de byte de S negativo
BXOR	H#D8	XOR de byte de R com S
BAND	H#E8	AND de byte de R com S
BOR	H#F8	OR de byte de R com S
Grupo 4		**Função**
CRC	H#00	Acumula caractere com redundância cíclica
SEL	H#10	Seleciona R ou S
SNORM	H#20	Normalização de comprimento simples
DNORM	H#30	Normalização de comprimento duplo
DIVRF	H#40	Ajuste do resto de divisão
SDIVQF	H#50	Ajuste do quociente de divisão com sinal
SMULI	H#60	Iteração de multiplicação com sinal
SMULT	H#70	Término de multiplicação com sinal
SDIVIN	H#80	Inicialização de divisão com sinal
SDIVIS	H#90	Início de divisão com sinal
SDIVI	H#A0	Iteração de divisão com sinal
UDIVIS	H#B0	Início de divisão sem sinal
UDIVI	H#C0	Iteração de divisão sem sinal
UMULI	H#D0	Iteração de multiplicação sem sinal
SDIVIT	H#E0	Término de divisão com sinal
UDIVIT	H#F0	Término de divisão sem sinal
Grupo 5		**Função**
LOADFF	H#0F	Carrega flip-flops de divisão/BCD
CLR	H#1F	Limpa
DUMPFF	H#5F	Ativa saída dos flip-flops de divisão/BCD
BCDBIN	H#7F	BCD para binário
EX3BC	H#8F	Correção de excesso-de-3 de byte
EX3C	H#9F	Correção de excesso-de-3 de palavra
SDIVO	H#AF	Teste de *overflow* de divisão sem sinal
BINEX3	H#DF	Binário para excesso-de-3
NOP32	H#FF	Nenhuma operação

Como exemplo de codificação usada para especificar os campos 17 a 27, considere a instrução para somar os conteúdos dos registradores 1 e 2 e colocar o resultado no registrador 3. A instrução simbólica é

```
CONT11 [17], WELH, SELRYFYMX, [24], R3, R2, R1, PASS+ADD
```

O montador traduz essa instrução no padrão de bits apropriado. Os componentes individuais da instrução podem ser descritos como a seguir:

- CONT11 é a instrução básica NOP (nenhuma operação).
- O campo [17] é alterado para WELH (habilita escrita, de bits de alta ordem e de baixa ordem), de modo que é efetuada escrita em um registrador de 32 bits.
- O campo [18] é alterado para SELRFYMX para selecionar retorno da saída do MUX da saída Y como entrada para a ULA.
- O campo [24] é alterado para designar o registrador R3 como registrador destino.
- O campo [25] é alterado para designar o registrador R2 como um dos registradores de entrada.
- O campo [26] é alterado para designar o registrador R1 como um dos registradores de entrada.
- O campo [27] é alterado para especificar a operação ADD (soma) para a ULA. A instrução do deslocador da ULA é PASS; portanto, a saída da ULA não sofre deslocamento.

Diversos pontos podem ser notados com relação à notação simbólica. Não é necessário especificar o número de campo para campos consecutivos. Isto é,

```
CONT11 [17], WELH, [18], SELRFYMX
```

pode ser escrito como:

```
CONT11 [17], WELH, SELRFYMX
```

porque SELRFYMX está no campo 18.

Instruções da ULA do Grupo 1 da Tabela 15.9 devem sempre ser usadas em conjunto com instruções do Grupo 2. Instruções dos Grupos 3 a 5 não devem ser usadas com instruções do Grupo 2.

15.5 APLICAÇÕES DE MICROPROGRAMAÇÃO

Desde a introdução da microprogramação, e especialmente desde o final da década de 60, as aplicações de microprogramação têm se tornado cada vez mais variadas e difundidas. Já em 1971, a maioria dos atuais usos de microprogramação estava em evidência (Flynn e Rosin, 1971). Artigos de revisão subseqüentes discutem, essencialmente, as mesmas aplicações (veja, por exemplo, Rauscher e Adams (1980)). O conjunto de aplicações atuais de microprogramação inclui as seguintes:

- Implementação de computadores
- Emulação
- Suporte para o sistema operacional
- Implementação de dispositivos de propósito especial

- Suporte para linguagem de alto nível
- Microdiagnóstico
- Adequação ao usuário

Este capítulo foi dedicado ao uso de microprogramação na *implementação de computadores*. A abordagem de microprogramação fornece uma técnica sistemática para implementação da unidade de controle. Uma técnica relacionada é a *emulação* (Mallach, 1975). Emulação refere-se ao uso de microprograma em uma máquina para executar programas originalmente escritos para outra máquina. O uso mais comum de emulação é auxiliar usuários a migrar de um computador para outro. Isso freqüentemente é usado pelo fabricante para tornar mais fácil aos seus clientes a mudança de máquinas antigas para máquinas novas, evitando que esses clientes considerem a mudança para uma máquina de outro fabricante mais atraente. Usuários freqüentemente se surpreendem em saber há quanto tempo esse artifício é utilizado. Um observador (Mallach e Sondak, 1983) notou que era ainda possível, em 1983, encontrar um IBM S/370 emulando um IBM 1401, que havia sido fisicamente substituído há mais de uma década e meia.

Outro uso proveitoso de microprogramação é na área de *suporte para sistema operacional*. Microprogramas podem ser usados para implementar primitivas que substituem partes importantes do software de sistema operacional. Essa técnica pode simplificar a implementação do sistema operacional e melhorar o seu desempenho.

A microprogramação é também útil como técnica para implementar *dispositivos de propósito especial*, que podem ser incorporados em um computador hospedeiro. Um bom exemplo é uma placa de comunicação de dados. Essa placa deve conter seu próprio microprocessador. Como ele está sendo usado para um propósito especial, faz sentido implementar algumas de suas funções em *firmware*, através de microprogramação, em lugar de implementar por software, para aumentar o desempenho.

Suporte para linguagem de alto nível é outra área de aplicação de técnicas de microprogramação. Várias funções e tipos de dados podem ser diretamente implementados em firmware. O resultado é que é mais fácil compilar programas em um código de máquina eficiente. De fato, a linguagem de máquina é adaptada às necessidades da linguagem de alto nível (por exemplo, FORTRAN, COBOL, Ada, C).

Microprogramação pode ainda ser usada para prover suporte para monitoração, detecção, isolamento e reparo de erros de sistema. Essas funções, conhecidas como *microdiagnóstico*, podem melhorar significativamente os recursos para manutenção do sistema. Essa abordagem possibilita ao sistema reconfigurar-se caso seja detectada uma falha; por exemplo, se um multiplicador de alta velocidade não está funcionando corretamente, um multiplicador multiprogramado pode substituí-lo.

Uma categoria genérica de aplicação é a *adequação ao usuário*. Diversas máquinas possuem uma memória de controle que pode ser alterada (isto é, uma memória de controle implementada em RAM e não em ROM) e permitem que o usuário escreva microprogramas. Geralmente, é provida uma microinstrução bastante vertical e fácil de usar. Isso possibilita ao usuário adequar a máquina à aplicação desejada.

15.6 LEITURAS RECOMENDADAS

Existem diversos livros dedicados à microprogramação. Talvez o mais didático seja Lynch (1993). Os fundamentos de codificação de microinstruções e o projeto de sistemas de

microcódigo são apresentados em Segee e Field (1991), por meio do projeto, passo a passo, de um processador simples de 16 bits. Carter (1996) também apresenta os conceitos básicos usando uma máquina simples. Uma descrição detalhada da Placa de Desenvolvimento de Software 8800 da Texas Instruments (TI 8800 SDB) pode ser encontrada em Parker e Hamblen (1989) e Texas Instruments Inc. (1990).

15.7 EXERCÍCIOS

15.1 Descreva a implementação da operação de multiplicação na máquina hipotética projetada por Wilkes. Use a narrativa e um fluxograma.

15.2 Considere um conjunto de microinstruções que inclui uma microinstrução com a seguinte forma simbólica:

```
IF (AC0 = 1) THEN CAR ← (C0-6) ELSE CAR ← (CAR) + 1
```

onde AC_0 é o bit de sinal do acumulador e $C_{0\text{-}6}$ são os primeiros sete bits da microinstrução. Usando essa microinstrução, escreva um microprograma para implementar a instrução de máquina BRM (*branch register minus*), que desvia se o conteúdo do acumulador for negativo. Suponha que os bits $C_1 - C_n$ da microinstrução especificam um conjunto paralelo de microoperações. Expresse o programa de forma simbólica.

15.3 O ciclo de instrução de um processador simples tem quatro fases principais: busca, indireção, execução e interrupção. Dois bits de estado designam a fase corrente em uma implementação por hardware.

a. Por que esses bits de estado são necessários?

b. Por que eles não são necessários em uma unidade de controle microprogramada?

15.4 Considere a unidade de controle da Figura 15.7. Suponha que a memória de controle tem largura de 24 bits. A parte de controle do formato da microinstrução é dividida em dois campos. Um campo de microoperação de 13 bits especifica a microoperação a ser efetuada. Um campo de seleção de endereço especifica uma condição, baseada nos bits de condição, que causa um desvio. Existem oito bits de condição.

a. Quantos bits tem o campo de seleção de endereço?

b. Quantos bits tem o campo de endereço?

c. Qual é o tamanho da memória de controle?

15.5 Como pode ser feito um desvio incondicional nas circunstâncias do exercício anterior? Como o desvio pode ser evitado? (Descreva uma microinstrução que não especifica nenhum desvio, condicional ou incondicional.)

15.6 Gostaríamos de prover 8 palavras de controle para cada rotina de instrução de máquina. Os códigos de operação de instruções de máquina têm 5 bits e a memória de controle tem tamanho de 1024 palavras. Sugira um mapeamento do registrador de instrução para o registrador de endereço de controle.

15.7 Considere um formato de microinstrução codificada. Mostre como um campo de microoperação de 9 bits pode ser dividido em subcampos para especificar 46 ações diferentes.

15.8 Um processador tem 16 registradores, uma ULA com 16 operações aritméticas e 16 operações lógicas, e uma unidade de deslocamento com 8 operações, todos conectados por um barramento de processador interno. Projete um formato de microinstrução para especificar as várias microoperações para o processador.

PARTE 5

ORGANIZAÇÃO PARALELA

OBJETIVOS

Esta parte final do livro trata de uma área de crescente importância, a área de organização paralela. Em uma organização paralela, diversas unidades de processamento cooperam entre si para executar uma aplicação. Enquanto um processador superescalar explora oportunidades de execução paralela no nível de instruções, uma organização de processamento paralelo busca explorar paralelismo em um nível mais alto, possibilitando que vários processadores trabalhem em paralelo e de forma cooperativa. Diversas questões são levantadas em relação a organizações desse tipo. Por exemplo, se diversos processadores, cada qual com sua memória cache individual, compartilham a mesma memória, devem ser empregados mecanismos de hardware ou de software que assegurem que todos os processadores compartilhem uma imagem válida da memória; isso é conhecido como problema de coerência de cache. Essa e outras questões de projeto são exploradas na Parte 5.

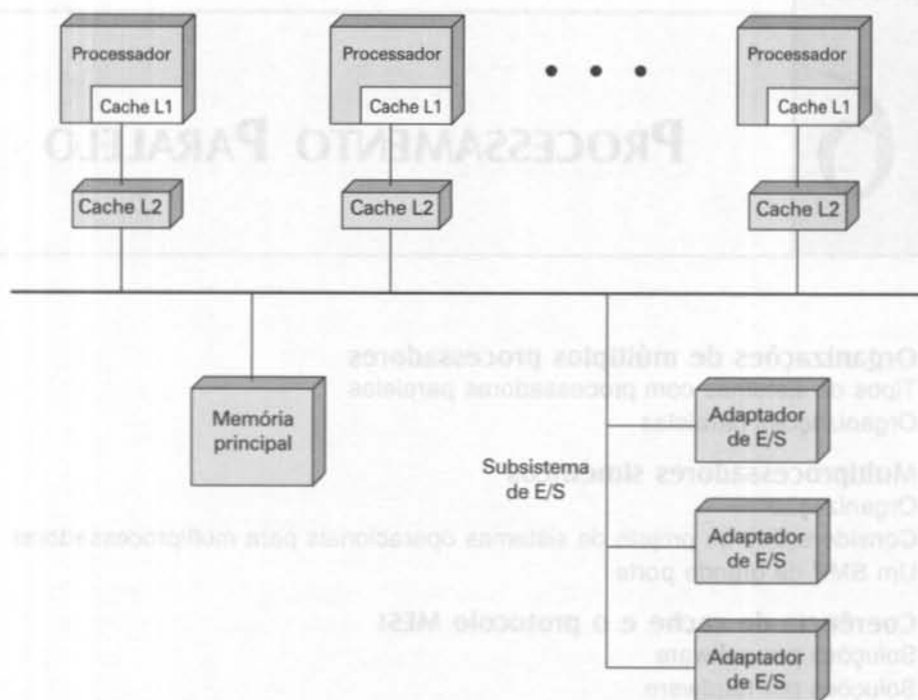
ROTEIRO

Capítulo 16 Processamento paralelo

O Capítulo 16 oferece uma visão geral sobre processamento paralelo. Em seguida, são descritas três abordagens para organização de múltiplos processadores: multiprocessadores simétricos (SMP), agregados (*clusters*) e máquinas com acesso não-uniforme à memória (NUMA). SMPs e *clusters* são as duas formas mais comuns de organizar vários processadores para obter melhor desempenho e disponibilidade. O conceito de sistemas NUMA é mais recente e seu uso comercial ainda não é muito difundido, mas mostra ser bastante promissor. Finalmente, o Capítulo 16 aborda uma organização especializada, conhecida como processador vetorial.

capítulo

16 PROCESSAMENTO PARALELO



- Uma maneira tradicional de aumentar o desempenho de um sistema de computador é usar vários processadores, que possam executar em paralelo para poder suportar uma dada carga de trabalho. As duas organizações de múltiplos processadores mais comuns são a de multiprocessadores simétricos (SMPs) e a de agregados (*clusters*). Mais recentemente, sistemas com acesso não-uniforme à memória (NUMA) têm sido introduzidos comercialmente.
- Uma organização SMP consiste em múltiplos processadores similares em um mesmo computador, conectados por um barramento ou alguma outra forma de circuito de conexão. O problema mais crítico nessa organização é o de coerência de cache. Cada processador possui sua própria memória cache e é possível que uma determinada linha de dados esteja presente em mais de uma cache. Se essa linha é alterada em uma das caches, então, tanto a memória principal como todas as demais caches terão uma versão inválida dessa linha. Protocolos de coerência de cache são projetados para tratar esse problema.
- Um cluster consiste de um conjunto de computadores completos, conectados entre si, que trabalham juntos como um recurso computacional unificado, criando a ilusão de ser uma única máquina. O termo *computador completo* (*whole computer*) é usado para designar um sistema que pode rodar por si próprio, independentemente do cluster.
- Um sistema NUMA consiste de um multiprocessador com memória compartilhada, no qual o tempo gasto por um certo processador para fazer acesso a uma palavra na memória varia de acordo com a posição dessa palavra na memória.

Tradicionalmente, o computador tem sido visto como uma máquina seqüencial. A maioria das linguagens de programação requer que o programador especifique um algoritmo como uma seqüência de instruções. Os processadores executam programas por meio da execução seqüencial de instruções de máquina. Cada instrução é executada como uma seqüência de operações (busca de instrução, busca de operandos, execução da operação, armazenamento de resultados).

Essa visão do computador nunca foi completamente verdadeira. No nível de microoperações, vários sinais de controle são gerados ao mesmo tempo. A técnica de *pipeline* de instruções tem sido usada há muito tempo, estendendo essa sobreposição pelo menos para as operações de busca e execução de instruções. Esses são dois exemplos de execução de funções em paralelo. Essa abordagem é levada mais adiante em uma organização superescalar, que explora paralelismo em nível de instrução. Nas máquinas superescalares, existem diversas unidades de execução em um mesmo processador, que podem assim executar várias instruções de um mesmo programa em paralelo.

À medida que a tecnologia evoluiu e o custo do hardware do computador tornou-se mais baixo, os projetistas de computadores têm buscado outras oportunidades de exploração de paralelismo, usualmente para melhorar o desempenho e, em alguns casos, para aumentar a disponibilidade do sistema. Depois de apresentar uma visão geral de organizações paralelas, este capítulo descreve três das abordagens mais importantes para organização paralela. Primeiramente, examinamos os multiprocessadores simétricos (SMP), uma das primeiras e ainda a mais comum dentre as organizações paralelas. Os SMPs usam múltiplos processadores compartilhando uma memória comum. A organização SMP levanta a questão de coerência de memórias cache, à qual é dedicada uma seção separada. Em seguida, descrevemos os agregados de computadores ou *clusters*, que consistem de diversos computadores independentes, organizados de forma cooperativa. Os clusters têm se tornado cada vez mais comuns, para suportar cargas de trabalho que estão além da capacidade de um único SMP. A terceira abordagem para o uso de múltiplos processadores é a de máquinas com acesso não-uniforme à memória (NUMA). A abordagem NUMA é relativamente nova e ainda não é muito usada comercialmente, mas tem sido considerada uma alternativa para as abordagens de clusters e SMPs. Finalmente, este capítulo examina as alternativas de organização de hardware para computação vetorial. Essas abordagens otimizam a ULA para o processamento de vetores de números de ponto flutuante. Elas têm sido usadas para implementar uma classe de sistemas conhecidos como *supercomputadores*.

16.1 ORGANIZAÇÕES DE MÚLTIPLOS PROCESSADORES

Tipos de sistemas com processadores paralelos

A taxonomia introduzida por Flynn (Flynn, 1972) é ainda a forma mais comum de classificar sistemas de processamento paralelo. Flynn propôs as seguintes categorias de sistemas de computação:

- **Única instrução, único dado** (SISD — *single instruction, single data*): um único processador executa uma única seqüência de instruções, usando dados armazenados em uma única memória. Um sistema uniprocessador pertence a essa categoria.
- **Única instrução, múltiplos dados** (SIMD — *single instruction, multiple data*): uma única instrução de máquina controla a execução simultânea de um certo número de ele-

mentos de processamento, em passos de execução. Cada elemento de processamento tem uma memória de dados a ele associada, de modo que cada instrução é executada sobre um conjunto de dados diferente em cada processador. Os processadores vetoriais e matriciais pertencem a essa categoria.

- **Múltiplas instruções, único dado** (MISD — *multiple instruction, single data*): uma seqüência de dados é transmitida para um conjunto de processadores, cada um dos quais executa uma seqüência de instruções diferente. Essa estrutura nunca foi implementada.
- **Múltiplas instruções, múltiplos dados** (MIMD): um conjunto de processadores executa simultaneamente seqüências diferentes de instruções, sobre conjuntos de dados distintos. Os SMPs, clusters e sistemas NUMA pertencem a essa categoria.

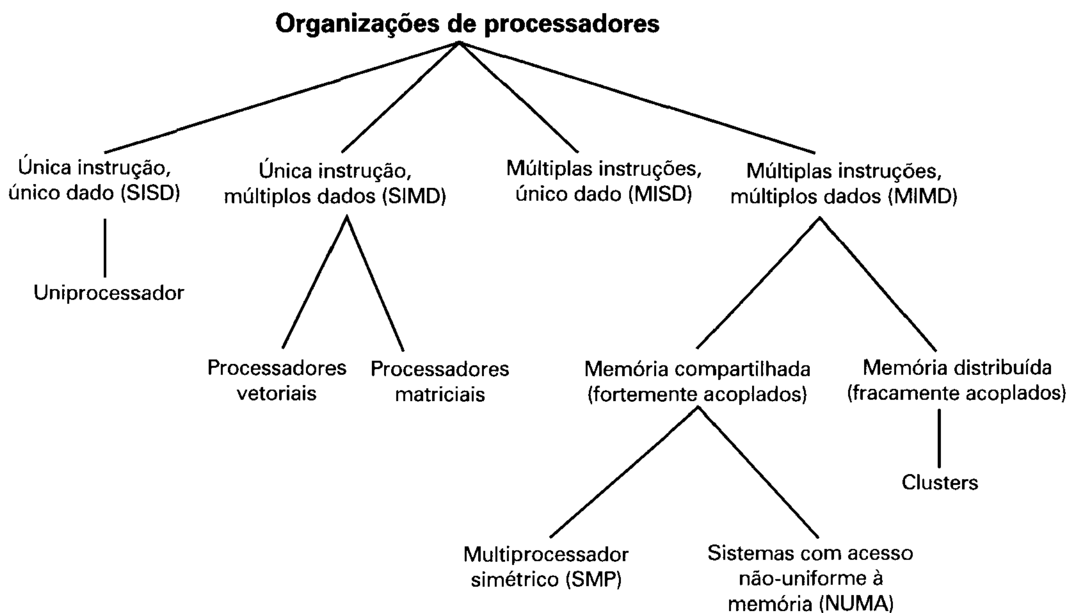


Figura 16.1 Uma taxonomia de arquiteturas com processadores paralelos.

Em uma organização MIMD, os processadores são de propósito geral; cada um pode processar todas as instruções necessárias para realizar a transformação de dados apropriada. Sistemas MIMD podem ser subdivididos de acordo com a forma de comunicação entre os processadores (Figura 16.1). Se os processadores compartilham uma memória comum, então cada processador usa programas e dados armazenados nessa memória compartilhada e se comunica com os outros processadores por meio dessa memória. A forma mais comum de sistemas desse tipo é conhecida como **multiprocessador simétrico** (SMP) e é examinada na Seção 16.2. Em um SMP, vários processadores compartilham uma única memória ou conjunto de memórias por meio de um barramento compartilhado ou de algum outro tipo de mecanismo de interconexão; uma característica particular desses sistemas é que o tempo de acesso a qualquer região da memória é aproximadamente o mesmo para cada processador. Um desenvolvimento mais recente é a organização com **acesso não-uniforme à memória** (NUMA), que é

descrita na Seção 16.5. Como o próprio nome sugere, o tempo de acesso a diferentes regiões da memória pode ser diferente para um processador NUMA.

Uma coleção de uniprocessadores ou de SMPs independentes pode ser conectada de modo que forme um **cluster**. A comunicação entre os computadores pode ser por meio de caminhos fixos ou de uma rede.

Organizações paralelas

A Figura 16.2 ilustra a organização geral de sistemas de cada categoria da taxonomia apresentada na Figura 16.1. A Figura 16.2a mostra a estrutura de uma máquina SISD. Existe uma unidade de controle (UC) que fornece uma seqüência de instruções (SI) para uma unidade de processamento (UP). A unidade de processamento opera sobre uma única seqüência de dados (SD) de uma única unidade de memória (UM). Em uma máquina SIMD, ainda existe uma única unidade de controle, que agora alimenta múltiplos elementos de processamento (EP) com uma única seqüência de instruções. Cada elemento de processamento tem sua própria memória (como ilustrado na Figura 16.2b) ou pode existir uma memória compartilhada. Finalmente, em uma organização MIMD, existem múltiplas unidades de controle, cada qual alimentando seu próprio elemento de processamento com uma seqüência de instruções diferente. Uma organização MIMD pode consistir de um multiprocessador com memória compartilhada (Figura 16.2c) ou pode ser um multicomputador com memória distribuída (Figura 16.2d).

As questões de projeto relativas a SMPs, clusters e sistemas NUMA são complexas, envolvendo aspectos de organização física, estruturas de interconexão, comunicação entre processadores, projeto do sistema operacional e técnicas de desenvolvimento de aplicações. Nosso principal objetivo é abordar o aspecto da organização, embora tratemos também, brevemente, sobre questões de projeto de sistemas operacionais.

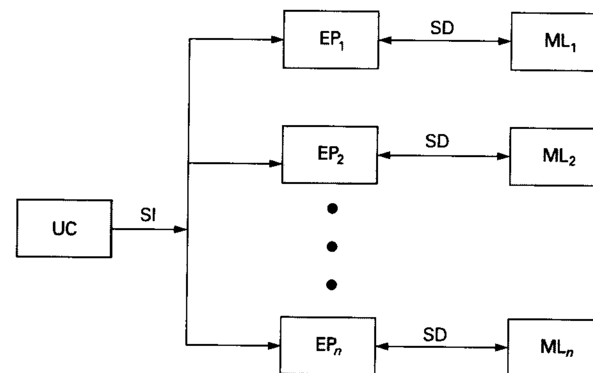
16.2 MULTIPROCESSADORES SIMÉTRICOS

Até muito recentemente, quase todos os computadores pessoais e a maioria das estações de trabalho continham um único microprocessador de uso geral. Com a crescente demanda por desempenho e a contínua queda do custo dos microprocessadores, os fabricantes introduziram sistemas SMP. A denominação SMP refere-se tanto à arquitetura de hardware do computador quanto ao comportamento do sistema operacional que reflete essa arquitetura. Um SMP pode ser definido como um sistema de computador independente, com as seguintes características:

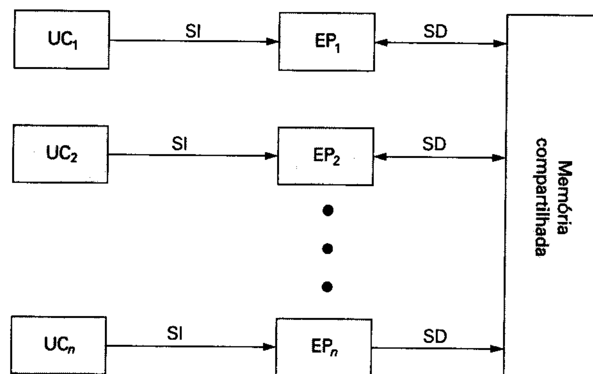
1. Existem dois ou mais processadores similares, com capacidade de computação comparável.
2. Esses processadores compartilham a mesma memória principal e facilidades de E/S e são conectados entre si por meio de um barramento ou de outro esquema de conexão interno, de forma que o tempo de acesso à memória é aproximadamente o mesmo para cada processador.
3. Todos os processadores compartilham acesso aos dispositivos de E/S, seja por meio de canais comuns ou de canais distintos que fornecem caminhos para os mesmos dispositivos.
4. Todos os processadores podem desempenhar as mesmas funções (daí o termo *simétrico*).
5. O sistema é controlado por um sistema operacional integrado, que provê interação entre os processadores e seus programas, em nível de tarefas, de arquivos e de dados.



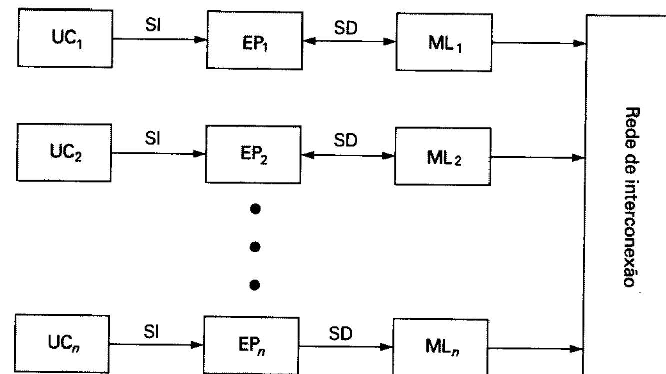
(a) SISD



(b) SIMD (com memória distribuída)



(c) MIMD (com memória compartilhada)



(d) MIMD (com memória distribuída)

Figura 16.2 Organizações alternativas de computadores.

Os itens 1 a 4 são auto-explicativos. O item 5 ilustra um dos contrastes com um sistema de multiprocessamento fracamente acoplado, tal como um cluster. Nesse último, a unidade física de interação é usualmente uma mensagem ou um arquivo inteiro. Em um SMP, elementos de dados individuais podem constituir o nível de interação, e pode haver um alto grau de cooperação entre os processadores.

O sistema operacional de um SMP efetua o escalonamento de processos ou fluxos de execução (ou *threads*) sobre todos os processadores. A arquitetura SMP tem uma série de vantagens potenciais sobre uma arquitetura uniprocessador:

- **Desempenho**: se o trabalho efetuado pelo computador pode ser organizado de forma que algumas porções desse trabalho possam ser feitas em paralelo, então um sistema com múltiplos processadores resulta em maior desempenho que um sistema do mesmo tipo com um único processador (Figura 16.3).
- **Disponibilidade**: em um multiprocessador simétrico, como todos os processadores são capazes de desempenhar as mesmas funções, uma falha em um único processador não causa a parada do sistema. Em vez disso, o sistema pode continuar a funcionar, com desempenho reduzido.
- **Crescimento incremental**: o usuário pode aumentar o desempenho do sistema adicionando novos processadores.
- **Escalabilidade**: fabricantes podem oferecer uma larga faixa de produtos, com características de desempenho e custo diferentes, com base no número de processadores configurados no sistema.

É importante notar que essas são vantagens potenciais, e não garantidas. O sistema operacional deve prover ferramentas e funções para explorar paralelismo em um sistema SMP.

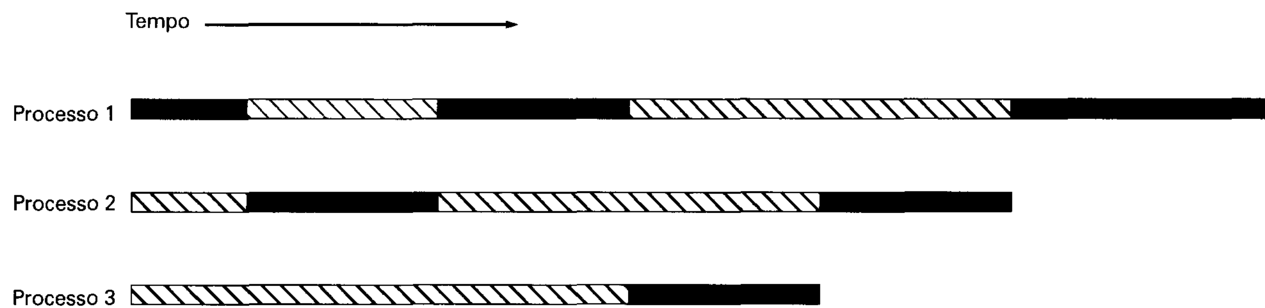
Uma característica atraente de um SMP é que a existência de múltiplos processadores é transparente para o usuário. O sistema operacional se encarrega do escalonamento de *threads* ou de processos nos processadores individuais e da sincronização entre processadores.

Organização

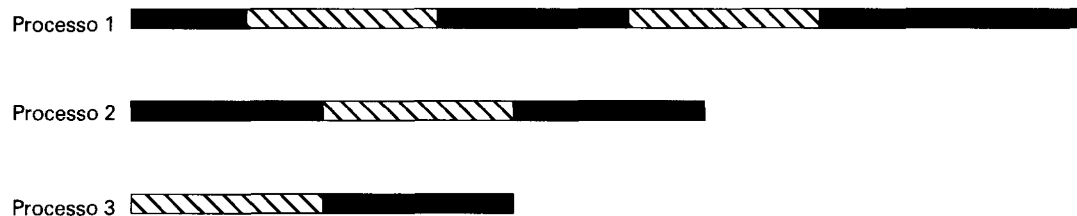
A Figura 16.4 mostra, em termos gerais, a organização de um sistema multiprocessador. Existem dois ou mais processadores. Cada processador é autocontido, incluindo uma unidade de controle, ULA, registradores e memória cache. Cada processador tem acesso a uma memória principal e dispositivos de E/S compartilhados, por meio de algum mecanismo de interconexão. Os processadores podem comunicar-se entre si por meio da memória (mensagens e informações de estado são armazenadas em áreas comuns). É também possível a troca direta de sinais entre os processadores. A memória freqüentemente é organizada de forma que podem ser efetuados acessos simultâneos a blocos separados de memória. Em algumas configurações, cada processador pode também ter, além dos recursos compartilhados, sua própria memória principal e canais de E/S privados.

A organização de um sistema multiprocessador pode ser classificada do seguinte modo:

- Tempo compartilhado ou barramento comum
- Memória com múltiplas portas
- Unidade de controle central



(a) Tempo compartilhado (multiprogramação)



(b) Tempo compartilhado e sobreposição (multiprocessamento)

Bloqueado Em execução

Figura 16.3 Multiprogramação e multiprocessamento.

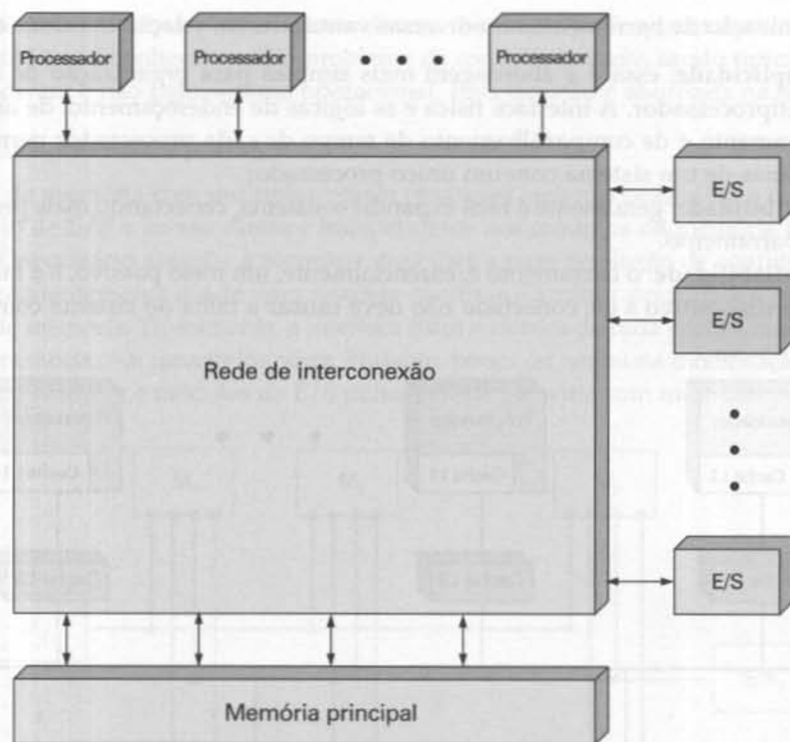


Figura 16.4 Diagrama de blocos geral de um multiprocessador fortemente acoplado.

Barramento de tempo compartilhado

O barramento de tempo compartilhado (*time-shared bus*) é o mecanismo mais simples de construção de um sistema multiprocessador (Figura 16.5). A estrutura e as interfaces são praticamente as mesmas de um sistema com um único processador que usa uma interconexão de barramento. O barramento consiste de linhas de controle, endereço e dados. Para facilitar transferências que usam DMA (acesso direto à memória) pelos processadores de E/S, são introduzidos os seguintes recursos:

- **Endereçamento**: deve ser possível distinguir módulos conectados ao barramento, para determinar a origem e o destino dos dados.
- **Arbitração**: qualquer módulo de E/S pode funcionar temporariamente como um "mestre" de barramento. É fornecido um mecanismo para arbitrar requisições de controle do barramento, usando algum esquema de prioridades.
- **Compartilhamento de tempo**: quando um módulo está controlando o barramento, ele fica bloqueado para uso pelos demais módulos, que devem necessariamente suspender sua operação até que o barramento seja liberado.

Essas características de um sistema uniprocessador podem ser empregadas diretamente em uma configuração SMP. Nesse caso, existirão múltiplos processadores, assim como múltiplos processadores de E/S, todos competindo por acesso a um ou mais módulos de memória, por meio do barramento.

A organização de barramento tem diversas vantagens em relação às outras abordagens:

- **Simplicidade**: essa é a abordagem mais simples para organização de um sistema multiprocessador. A interface física e as lógicas de endereçamento, de arbitração de barramento e de compartilhamento de tempo de cada processador permanecem as mesmas de um sistema com um único processador.
- **Flexibilidade**: geralmente é fácil expandir o sistema, conectando mais processadores ao barramento.
- **Confiabilidade**: o barramento é, essencialmente, um meio passivo, e a falha de qualquer dispositivo a ele conectado não deve causar a falha do sistema como um todo.

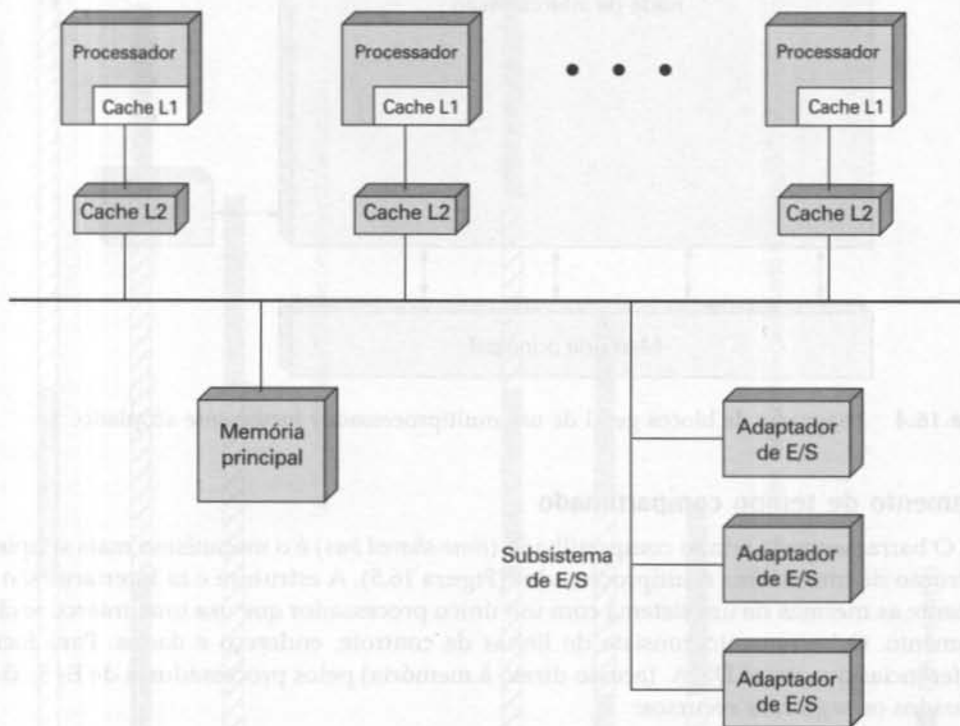


Figura 16.5 Organização de multiprocessador simétrico.

A principal desvantagem da organização de barramento é o desempenho. Toda referência à memória passa através do barramento comum. Portanto, a velocidade do sistema é limitada pelo tempo de ciclo do barramento. Para aumentar o desempenho, seria desejável equipar cada processador com uma memória cache. Isso reduziria dramaticamente o número de acessos ao barramento. Tipicamente, microcomputadores e estações de trabalho SMP possuem dois níveis de cache, sendo a cache L1 interna (contida na mesma pastilha do processador) e a cache L2 interna ou externa.

O uso de memórias cache introduz algumas novas considerações de projeto. Como cada cache local contém uma imagem de uma porção da memória, a alteração de uma palavra em uma memória cache pode, possivelmente, invalidar cópias dessa palavra em outras caches.

Para prevenir isso, os outros processadores devem ser alertados de que uma atualização ocorreu. Esse problema é conhecido como problema de *coerência de cache*, sendo tipicamente tratado pelo hardware e não pelo sistema operacional. Essa questão é abordada na Seção 16.3.

Memória com múltiplas portas

O uso de memória com múltiplas portas (*multiport memory*) possibilita a cada processador e módulo de E/S o acesso direto e independente aos módulos da memória principal (Figura 16.6). É necessário associar à memória uma lógica para resolução de conflitos de acesso. O método freqüentemente usado para resolver conflitos é atribuir prioridades permanentes a cada porta de memória. Tipicamente, a interface física e elétrica de cada porta é idêntica à de um módulo de memória com uma única porta. Portanto, pouca ou nenhuma modificação é requerida para que processadores e módulos de E/S utilizem uma memória com múltiplas portas.

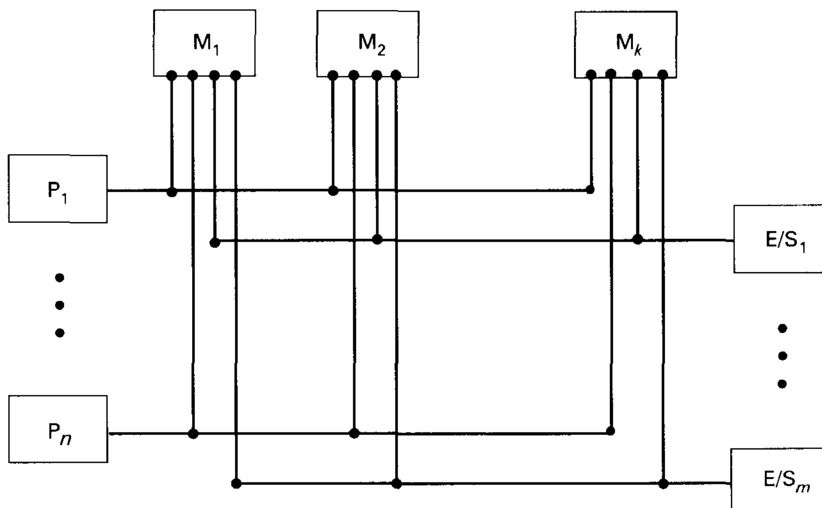


Figura 16.6 Memória com múltiplas portas.

A abordagem de memória de múltiplas portas é mais complexa que a abordagem de barramento compartilhado, requerendo a adição de uma grande quantidade de circuitos lógicos ao sistema de memória. Essa abordagem deve, entretanto, fornecer melhor desempenho, pois cada processador tem um caminho dedicado para cada módulo de memória. Outra vantagem da memória de múltiplas portas é que é possível configurar porções da memória como 'dedicadas' para uso de um ou mais processadores e/ou módulos de E/S. Essa característica oferece maior segurança contra acessos não-autorizados, assim como o armazenamento de rotinas de recuperação do sistema em áreas de memória não-suscetíveis de modificação por outros processadores.

Outro ponto é que deve ser usada uma política de escrita direta para controle da cache, pois não existe uma forma conveniente de alertar outros processadores sobre uma alteração na memória.

Unidade de controle central

A unidade de controle central comanda fluxos de dados distintos de e para módulos independentes: processadores, memória, E/S. O controlador pode armazenar temporariamente requisições e desempenhar funções de arbitração e temporização. Ele pode também passar mensagens de controle e de estado entre os processadores e alertar processadores quando uma memória cache é atualizada.

Como a lógica para coordenar a configuração de múltiplos processadores está toda concentrada na unidade de controle central, as interfaces de E/S, de processadores e da memória permanecem essencialmente inalteradas. Isso resulta na flexibilidade e na simplicidade de interfaceamento da abordagem de barramento. As principais desvantagens dessa abordagem são que a unidade de controle central é muito complexa e é um potencial gargalo no desempenho do sistema.

A estrutura de unidade de controle central foi bastante comum em sistemas de grande porte com múltiplos processadores, tais como membros de maior porte da família IBM S/370. Essa estrutura é raramente empregada hoje em dia.

Considerações de projeto de sistemas operacionais para multiprocessadores

Um sistema operacional SMP gerencia o uso dos processadores e demais recursos do computador de maneira que o usuário perceba um único sistema operacional controlando os recursos do sistema. De fato, essa configuração deve funcionar para um usuário como um sistema multiprogramado com um único processador. Tanto no caso do SMP como no de um sistema uniprocessador, podem existir várias tarefas ou processos ativos ao mesmo tempo, e é responsabilidade do sistema operacional escalonar suas execuções e alocar recursos. Um usuário pode desenvolver aplicações que usam vários processos ou vários *threads* em um processo, sem se preocupar se existe um único processador ou diversos processadores disponíveis. Portanto, um sistema operacional multiprocessador deve prover todas as funcionalidades de um sistema multiprogramado, além de recursos adicionais para acomodar múltiplos processadores. Entre as principais questões de projeto, estão as seguintes:

- **Processos concorrentes simultâneos**: as rotinas do sistema operacional devem ser reentrantes, para permitir que vários processadores executem um mesmo código do sistema operacional simultaneamente. Vários processadores podem estar executando a mesma ou partes diferentes do sistema operacional, assim as tabelas e estruturas de gerenciamento do sistema operacional devem ser gerenciadas de forma adequada, para evitar situações de impasse (bloqueio perpétuo ou *deadlock*) ou operações inválidas.
- **Escalonamento**: o escalonamento pode ser feito por qualquer processador, sendo necessário evitar conflitos. O escalonador deve atribuir processos prontos a processadores disponíveis.
- **Sincronização**: como podem existir vários processos ativos com acesso potencial a espaços de endereçamento e recursos de E/S compartilhados, é necessário prover uma sincronização efetiva. A sincronização é um mecanismo que força exclusão mútua e ordenação de eventos.
- **Gerenciamento de memória**: o gerenciamento de memória de um multiprocessador deve lidar com todos os aspectos encontrados nos sistemas com um único processa-

dor, como discutido no Capítulo 7. Além disso, o sistema operacional deve explorar o paralelismo disponível no hardware, como, por exemplo, memória com múltiplas portas, para obter maior desempenho. Os mecanismos de paginação dos diferentes processadores devem ser coordenados, para a garantir coerência entre os diversos processadores que compartilhem uma página ou segmento e decidir, quando necessário, quais páginas devem ser substituídas.

- **Confiabilidade e tolerância a falhas**: o sistema operacional deve prover recursos para que haja uma degradação suave do sistema em caso de falhas de processadores. O escalonador, assim como outras partes do sistema operacional, deve reconhecer a perda de um processador e reestruturar as tabelas de gerenciamento de acordo.

Um SMP de grande porte

A maioria dos microcomputadores e estações de trabalho SMP usa a estratégia de interconexão por meio de barramento, como mostrado na Figura 16.5. É instrutivo examinar uma abordagem alternativa, usada nas implementações mais recentes da família de computadores de grande porte (mainframe) IBM S/390 (Mak et al, 1997).

A Figura 16.7 mostra a organização global do S/390 SMP. Essa família de sistemas abrange desde um sistema uniprocessador com um único cartão de memória até um sistema com dez processadores e quatro cartões de memória. A configuração inclui também um ou dois processadores adicionais, que servem como processadores de E/S. Os principais componentes da configuração são:

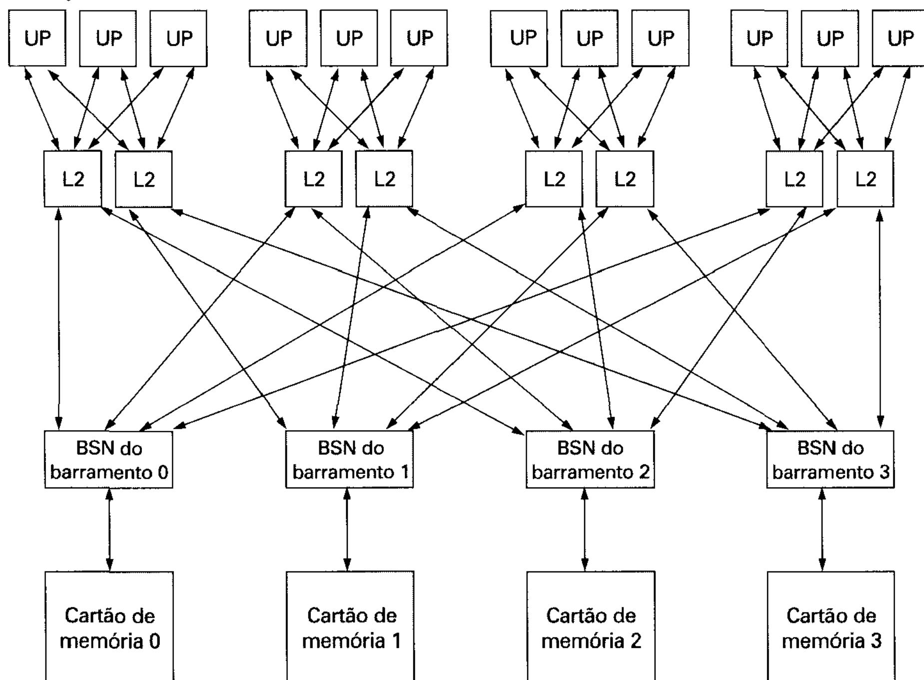


Figura 16.7 Organização SMP do IBM S/390.

- **Unidade de processamento (UP)**: é um microprocessador CISC, no qual as instruções usadas mais freqüentemente são implementadas em hardware e as demais são executadas por firmware. Cada unidade de processamento inclui uma cache L1 de 64 KB, unificada (para instruções e dados). O tamanho da cache L1 foi escolhido de modo que pudesse ser acomodada na pastilha da unidade de processamento e que o tempo de acesso à cache fosse igual a um ciclo de processador.
- **Cache L2**: cada cache L2 contém 384 KB. As caches L2 são arranjadas em grupos de duas, com cada grupo possibilitando conectar três unidades de processamento e permitindo o acesso a todo o espaço de memória principal.
- **Adaptador de rede de comutação de barramentos (BSN)**: os BSNs interconectam as caches L2 e a memória principal. Cada BSN inclui uma cache de nível 3 (L3), com tamanho de 2 MB.
- **Cartão de memória**: Cada cartão contém 8 GB de memória, sendo a capacidade total de memória de 32 GB.

A configuração SMP do S/390 tem diversas características interessantes, discutidas a seguir:

- Interconexão chaveada
- Caches L2 compartilhadas
- Cache L3

Interconexão chaveada

O uso de um único barramento compartilhado é comum em organizações SMP de microcomputadores e estações de trabalho (Figura 16.5). Nesse arranjo, o barramento único se torna um gargalo do sistema, afetando a possibilidade de expansão do sistema (escalabilidade). Esse problema é resolvido no S/390 de duas formas. Primeiro, a memória principal é dividida em quatro cartões de memória separadas, cada qual com seu próprio controlador, que pode tratar acessos à memória com alta velocidade. A carga média de tráfego para a memória é reduzida por um fator igual a quatro, devido à existência de quatro caminhos independentes para as quatro partes separadas da memória. Em segundo lugar, a conexão dos processadores (na verdade, das caches L2) a um único cartão de memória não é na forma de um barramento compartilhado, mas sim por meio de conexões ponto a ponto, onde cada conexão liga um grupo de processadores, via uma cache L2, a um BSN. O BSN, por sua vez, desempenha uma função de comutação, roteando dados entre suas cinco conexões (quatro conexões com caches L2 e uma conexão com cartão de memória). Com relação às quatro conexões com caches L2, o BSN conecta quatro ligações físicas a um barramento lógico de dados. Assim, um sinal de entrada em qualquer das quatro conexões com caches L2 é ecoado de volta para as demais conexões com caches L2; essa característica é requerida para prover suporte a coerência de cache.

Note que, embora existam quatro cartões de memória separados, cada unidade de processamento e cada cache L2 tem apenas duas portas físicas em direção à memória principal. Isso porque cada cache L2 armazena apenas dados de metade da memória principal. É necessário um par de caches para servir toda a memória principal, e cada unidade de processamento deve ser conectada a ambas as caches de um par.

Caches L2 compartilhadas

Em um sistema SMP típico com dois níveis de cache, cada processador possui uma cache L1 dedicada e uma cache L2, também dedicada. Mais recentemente, tem crescido o uso do conceito de cache L2 compartilhada. Nas primeiras versões de seu S/390 SMP, conhecidas como geração 3 (G3), a IBM fazia uso de caches L2 dedicadas. Nas versões mais recentes (G4 e G5), é usada uma cache L2 compartilhada. Duas considerações ditaram essa mudança:

1. Do G3 para o G4, a IBM dobrou a velocidade dos microprocessadores. Se fosse mantida a organização do G3, ocorreria um aumento significativo de tráfego no barramento. Ao mesmo tempo, era desejável reutilizar o maior número possível de componentes do G3. Sem uma melhora significativa do barramento, os BSNs se tornariam um gargalo do sistema.
2. Análises de cargas de trabalho típicas do IBM S/390 revelaram um alto grau de compartilhamento de instruções e dados entre os processadores.

Essas considerações levaram a equipe de projeto do S/390 G4 a usar uma ou mais caches L2, cada uma compartilhada entre múltiplos processadores (cada processador possuindo uma cache dedicada L1 na própria pastilha). À primeira vista, o compartilhamento da cache L2 pode parecer uma má idéia. O acesso à memória pelos processadores deveria tornar-se mais lento, porque cada processador teria de competir por acesso a uma única cache L2. Entretanto, se realmente for compartilhada uma quantidade suficiente de dados entre os múltiplos processadores, o uso de uma cache compartilhada poderá aumentar, e não diminuir, a taxa de acesso a dados. Os dados compartilhados poderiam ser obtidos mais rapidamente na cache compartilhada, do que seriam obtidos por meio do barramento.

Tabela 16.1 Taxa de acerto na cache típica na configuração S/390 SMP

Subsistema de memória	Penalidade de acesso (ciclos da UP)	Tamanho da cache	Taxa de acerto (%)
Cache L1	1	32 KB	89
Cache L2	5	256 KB	5
Cache L3	14	2 MB	3
Memória	32	8 GB	3

Uma abordagem considerada pela equipe de projeto do S/390 G4 foi o uso de uma grande cache única, compartilhada por todos os processadores. Embora isso pudesse melhorar o desempenho devido à maior eficiência da cache, essa abordagem teria requerido um completo reprojeto da organização de barramento existente. Análises de desempenho indicaram que a introdução de compartilhamento de cache em cada um dos barramentos BSN teria a maioria das vantagens de caches compartilhadas, reduzindo, ao mesmo tempo, o tráfego no barramento. A utilidade das caches compartilhadas foi confirmada por medidas de desempenho que mostraram que o uso delas aumentava significativamente as taxas de acerto nas caches, em comparação com o esquema de caches dedicadas usado na organização G3 (Mak et al, 1997). Estudos sobre a utilidade de caches compartilhadas em configurações SMP de menor porte confirmaram o valor dessa abordagem (Nayfeh, Olukotun e Singh, 1996).

Cache L3

Outra característica interessante do S/390 SMP é o uso de um terceiro nível de cache (L3).[1] As caches L3 são localizadas nos BSNs e, portanto, cada cache L3 provê uma área de armazenamento temporário entre uma cache L2 e um cartão de memória. A cache L3 reduz a latência para acesso a dados que não estejam presentes nas caches L1 e L2 do processador que requisitou o dado. Ela fornece o dado muito mais rapidamente que a memória principal, caso a linha de cache requerida já seja compartilhada por outros processadores, mas não tenha sido usada recentemente pelo processador que requisitou o dado.

A Tabela 16.1 mostra resultados de desempenho desse sistema SMP com três níveis de memória cache, para cargas de trabalho constituídas de aplicações comerciais típicas do S/390, com alta carga de acesso ao barramento e à memória (Doetting et al, 1997).[2] A penalidade de acesso à memória é o tempo de latência existente entre o instante em que um dado é requisitado à hierarquia de cache e aquele em que o primeiro bloco de dados de 16 bytes é recebido. A cache L1 tem uma taxa de acerto de 89%; portanto, apenas 11% das referências à memória devem ser resolvidos nas caches L2 e L3 ou na memória principal. Desses 11%, 5% são resolvidos no nível L2, e assim por diante. Com o uso de três níveis de cache, apenas 3% das referências requerem acesso à memória principal. Sem o terceiro nível, a taxa de acesso à memória principal dobra.

16.3 COERÊNCIA DE CACHE E O PROTOCOLO MESI

Nos sistemas multiprocessadores modernos, é comum o uso de um ou dois níveis de cache associados a cada processador. Essa organização é essencial para obter desempenho razoável. Entretanto, ela cria um problema conhecido como problema de *coerência de cache*. A essência desse problema é a seguinte: podem existir simultaneamente múltiplas cópias do mesmo dado em diferentes caches; se os processadores puderem atualizar suas cópias livremente, o resultado será uma imagem da memória incoerente. No Capítulo 4, definimos duas políticas comuns de escrita na cache:

- **Escrita de volta (*write back*)**: as operações de escrita usualmente são efetuadas apenas sobre a cache. A memória principal somente é atualizada quando a linha correspondente é removida da cache.
- **Escrita direta (*write through*)**: toda operação de escrita é feita na memória principal, assim como na cache, assegurando que a memória principal sempre seja mantida válida.

É claro que a política de escrita de volta pode resultar em incoerência. Se duas caches contêm a mesma linha, e essa linha é atualizada em uma das caches, a outra cache não saberá que contém um dado inválido. Leituras subseqüentes dessa linha inválida produzem resultados inválidos. Mesmo com a política de escrita direta pode ocorrer incoerência, a não ser que

1 A literatura da IBM refere-se a essa cache como uma cache L2,5. Parece não haver nenhuma vantagem particular no uso desse termo, pois, de fato, essa cache constitui um terceiro nível de cache.

2 Esses dados são para um sistema G3, que usa caches L2 dedicadas. Entretanto, os resultados sugerem o desempenho esperado com caches L2 compartilhadas, tal como encontrado nos sistemas S/390 G4 e G5.

a outra cache monitore o tráfego de memória ou receba diretamente alguma notificação direta da atualização.

Nesta seção, apresentamos uma breve visão geral das várias abordagens usadas para resolver o problema de coerência de cache, dando maior enfoque à abordagem que é mais utilizada: o protocolo MESI. Uma versão desse protocolo é usada nas implementações do Pentium II e do PowerPC.

O objetivo de qualquer protocolo de coerência de cache é permitir que variáveis locais usadas mais recentemente sejam armazenadas na cache apropriada e permaneçam lá por várias operações de leitura e escrita, enquanto o protocolo é usado para manter a coerência de variáveis compartilhadas, que podem estar presentes em múltiplas caches simultaneamente. As abordagens para coerência de cache geralmente são divididas em abordagens por software ou por hardware. Algumas implementações adotam uma estratégia que envolve elementos tanto de hardware como de software. Mesmo assim, a classificação em abordagens por software ou por hardware permanece instrutiva e é usada comumente nas descrições de estratégias de coerência de cache.

Soluções por software

Os esquemas de coerência de cache implementados por software procuram evitar a necessidade de circuitos e lógica adicional no hardware, confiando no compilador e no sistema operacional para tratar o problema. As abordagens por software são atrativas porque a sobrecarga de detectar potenciais problemas em uma aplicação é transferida do tempo de execução para o tempo de compilação, e a complexidade de projeto é transferida do hardware para o software. Por outro lado, abordagens por software, executadas em tempo de compilação, geralmente devem tomar decisões conservativas, levando a uma utilização ineficiente da cache.

Mecanismos de coerência de cache baseados em compilador efetuam uma análise do código de programas, para distinguir dados que podem causar problema de coerência se armazenados na cache, marcando esses dados. O sistema operacional ou o hardware evitam que esses dados sejam armazenados na cache.

A abordagem mais simples é evitar que variáveis compartilhadas sejam armazenadas na cache. Essa estratégia é excessivamente conservativa, pois estruturas de dados compartilhadas podem ser usadas de forma exclusiva durante alguns períodos de tempo e podem ser usadas apenas para leitura em outros períodos. O problema de coerência ocorre apenas quando ao menos um processador pode alterar a variável e outro processador pode usar essa variável.

Abordagens mais eficientes analisam o código buscando determinar períodos em que variáveis compartilhadas podem ser armazenadas na cache de forma segura. O compilador então insere instruções no código gerado para assegurar a coerência de caches nos períodos críticos. Foram desenvolvidas diversas técnicas para efetuar a análise e garantir os resultados; veja, por exemplo, (Lilja, 1993) e (Stenstrom, 1990).

Soluções por hardware

As soluções baseadas em hardware são geralmente denominadas protocolos de coerência de cache. Essas soluções possibilitam reconhecimento dinâmico, em tempo de execução, de potenciais condições para incoerências de cache. Como o problema é tratado apenas quando ele realmente ocorre, tem-se um uso mais efetivo das caches, resultando em melhor desempenho

que no caso das soluções por software. Além disso, essas abordagens são transparentes para o programador e para o compilador, reduzindo as dificuldades de desenvolvimento de software.

Os esquemas baseados em hardware diferem entre si em diversas particularidades, incluindo onde é mantida a informação sobre o estado de linhas de dados, como essa informação é organizada, onde é garantida a coerência e quais os mecanismos para garantir coerência. De maneira geral, os esquemas baseados em hardware podem ser divididos em duas categorias: protocolos de diretório e protocolos de monitoração (*snoopy protocols*).

Protocolos de diretório

Os protocolos de diretório coletam e mantêm informação sobre a localização de cópias de linhas de dados. Tipicamente, existe um controlador centralizado, que é parte do controlador da memória principal, e um diretório que é armazenado na memória principal. Esse diretório contém a informação de estado global sobre o conteúdo das várias caches locais. Quando um controlador de cache individual efetua uma requisição, o controlador centralizado verifica essa requisição e envia os comandos necessários para transferência de dados entre a memória e uma cache ou entre as próprias caches. O controlador central é também responsável por manter a informação de estado atualizada; portanto, toda ação local que possa afetar o estado global de uma linha deve ser comunicada ao controlador central.

Tipicamente, o controlador mantém informação sobre quais processadores possuem cópias de quais linhas. Antes que um processador possa atualizar sua cópia local de uma linha, ele deve requisitar ao controlador acesso exclusivo a essa linha. Para ceder esse acesso exclusivo à linha, primeiro o controlador envia uma mensagem a todos os processadores com uma cópia cacheada dessa linha, forçando cada processador a invalidar sua cópia. Depois de receber uma mensagem de confirmação de cada processador, o controlador cede o acesso exclusivo ao processador requisitante. Quando um processador tenta ler uma linha que está cedida com exclusividade a outro processador, ele envia uma notificação de falha (*miss*) de acesso para o controlador. O controlador então envia um comando para o processador que detém a linha, requerendo que ele escreva essa linha de volta na memória principal. A linha pode então ser compartilhada para leitura pelo processador original e pelo processador requisitante.

Esquemas baseados em diretório têm a desvantagem de introduzir um gargalo central e da sobrecarga de comunicação entre os vários controladores de cache e o controlador central. Entretanto, eles são eficazes em sistemas de grande escala, envolvendo múltiplos barramentos ou algum outro esquema de interconexão mais complexo.

Protocolos de monitoração

Protocolos de monitoração (*snoopy protocols*) distribuem a responsabilidade de manter coerência de cache entre todos os controladores de cache em um multiprocessador. Uma cache deve reconhecer quando uma linha, que ela armazena, é compartilhada com outras caches. Quando uma linha de cache compartilhada é alterada, isso deve ser comunicado a todas as demais caches, por meio de um mecanismo de difusão (*broadcast*). Cada controlador de cache é capaz de monitorar a rede para detectar essas notificações difundidas e reagir de forma apropriada.

Os protocolos de monitoração idealmente são adequados para multiprocessadores baseados em barramento, porque um barramento compartilhado provê um mecanismo simples

para difusão de mensagens e monitoramento. Entretanto, como um dos objetivos do uso de caches locais é evitar o acesso ao barramento, deve-se ter cuidado para que o aumento de tráfego no barramento requerido para difusão e monitoramento não cancele o ganho obtido com o uso de caches locais.

Duas abordagens básicas de protocolo de monitoração têm sido exploradas: escrita com invalidação e escrita com atualização (ou escrita em difusão). Em um protocolo de escrita com invalidação (*write-invalidate*), pode haver múltiplos leitores mas apenas um escritor, em qualquer instante. Inicialmente, uma linha pode ser compartilhada entre várias caches para fins de leitura. Quando uma das caches quer escrever sobre a linha, ela primeiro envia uma mensagem que invalida essa linha nas demais caches, o que torna a linha exclusiva para a cache que irá efetuar a escrita. Uma vez que a linha seja exclusiva, o processador que detém a linha pode efetuar operações de escrita locais e baratas, até que algum outro processador requisite a mesma linha.

Em um protocolo de escrita com atualização (*write-update*), podem existir múltiplos escritores e múltiplos leitores ao mesmo tempo. Quando um processador deseja escrever sobre uma linha compartilhada, a palavra a ser atualizada é distribuída para todas as outras caches, de modo que aquelas que contêm essa palavra possam também atualizá-la.

Nenhuma dessas abordagens é superior à outra em qualquer circunstância. O desempenho depende do número de caches locais e do padrão das leituras e escritas à memória. Alguns sistemas implementam protocolos adaptativos, que empregam tanto o mecanismo de escrita com invalidação quanto o mecanismo de escrita com atualização.

A abordagem de escrita com invalidação é a mais usada em sistemas de multiprocessadores comerciais, por exemplo, no Pentium II e no PowerPC. O estado de cada linha é marcado (usando dois bits extras no rótulo da cache) como modificado, exclusivo, compartilhado ou inválido. Da designação dada a esses estados em inglês, isto é, *Modified*, *Exclusive*, *Shared* e *Invalid*, deriva a denominação MESI, dada ao protocolo de escrita com invalidação. O protocolo MESI já foi abordado no Capítulo 4, quando lidamos com a questão de coordenação de caches locais de nível 1 e de nível 2. No restante desta seção, examinamos o uso desse protocolo entre caches locais de um sistema multiprocessador. Para simplificar a apresentação, não examinamos os mecanismos envolvidos na coordenação de caches de nível 1 e de nível 2, seja localmente seja entre múltiplos processadores distribuídos. Isso apenas complicaria a discussão, não introduzindo nenhum princípio novo.

O protocolo MESI

Você deve lembrar-se do Capítulo 4 que, com o protocolo MESI, a cache de dados inclui dois bits extras no rótulo, de modo que cada linha pode estar em um dos quatro estados seguintes:

- **Modificada**: a linha da cache foi modificada (é diferente da memória principal) e está presente apenas nessa cache.
- **Exclusiva**: a linha da cache é igual àquela na memória principal e não está presente em nenhuma outra cache.
- **Compartilhada**: a linha da cache é igual àquela na memória principal e pode estar presente em outra cache.
- **Inválida**: a linha da cache não contém dados válidos.

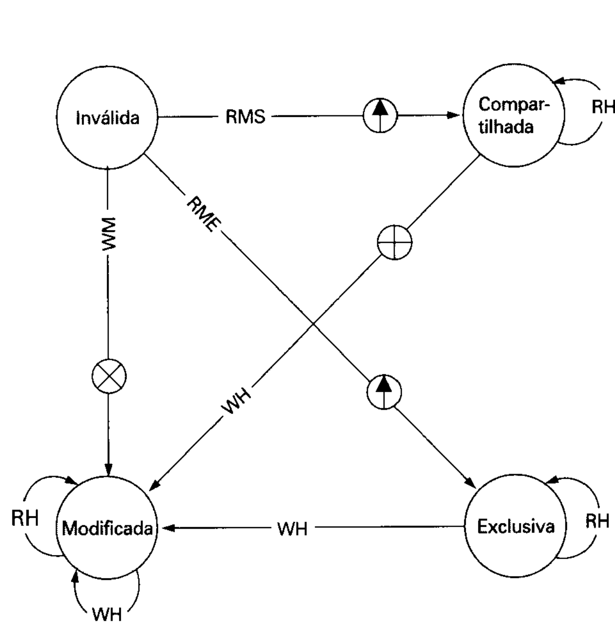
A Figura 16.8 mostra o diagrama de transição de estados para o protocolo MESI. Lembre-se de que cada linha da cache tem seus próprios bits de estado e, portanto, sua própria instância do diagrama de transição de estados. A Figura 16.8a mostra as transições que ocorrem devido a ações iniciadas pelo processador ao qual a cache está associada. A Figura 16.8b mostra as transições que ocorrem devido a eventos que são monitorados no barramento comum. Essa apresentação de ações iniciadas pelo processador e ações iniciadas pelo barramento em diagramas de transição de estados separados ajuda a esclarecer a lógica do protocolo MESI. Entretanto, em qualquer instante, a linha da cache está em um único estado. Se o próximo evento é originado pelo processador associado, a transição é ditada pela Figura 16.8a; se o próximo evento é originado pelo barramento, a transição é ditada pela Figura 16.8b. Examinemos essas transições mais detalhadamente.

Falha na leitura

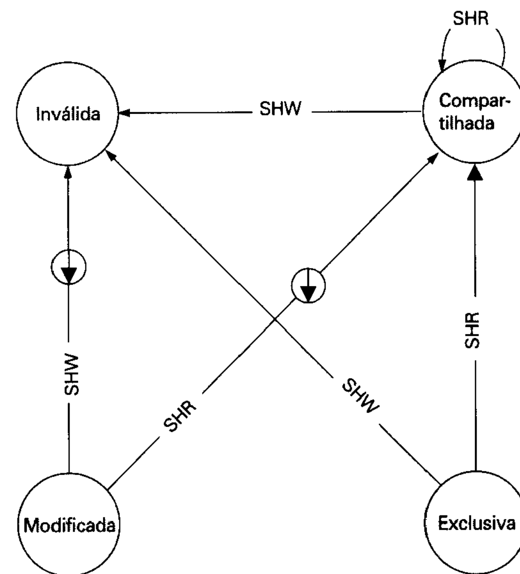
Quando ocorre uma falha na leitura (*read miss*) na sua cache local, o processador inicia uma leitura na memória para buscar a linha que contém o endereço da falha. O processador insere um sinal no barramento, que alerta todas as demais unidades processador/cache para monitorar a transação. Existem vários resultados possíveis:

- Se uma outra cache tem uma cópia limpa (não modificada desde que foi lida da memória principal) da linha requerida, em estado exclusivo, ela retorna um sinal indicando que compartilha essa linha. O processador que enviou essa resposta altera o estado da linha da cache, de exclusiva para compartilhada, e o processo que iniciou a transação lê a linha na memória principal e altera o estado da sua cópia da linha de inválida para compartilhada.
- Se uma ou mais caches possuem uma cópia limpa da linha em estado compartilhado, cada uma delas sinaliza que compartilha essa linha. O processo que iniciou a transação lê a linha e altera o estado da sua cópia de inválida para compartilhada.
- Se uma outra cache tem uma cópia modificada da linha requerida, então essa cache bloqueia a leitura na memória e provê a linha para a cache requisitante por meio do barramento compartilhado. A cache que forneceu a linha altera então o estado da sua cópia da linha de modificada para compartilhada.[3]
- Se nenhuma cache possui uma cópia da linha (limpa ou modificada), então nenhum sinal é retornado ao barramento. O processador que iniciou a transação então lê a linha requerida na memória principal e altera o estado da sua linha de inválida para exclusiva.

3 Em algumas implementações, a cache que possui uma cópia modificada da linha envia ao processador que iniciou a transação um sinal para que ele tente a operação novamente. Enquanto isso, o processador que possui a cópia modificada obtém o controle do barramento, escreve a linha modificada de volta na memória principal e altera o estado da linha na sua cache, de modificada para compartilhada. Subseqüentemente, o processador requisitante tenta nova- mente obter uma cópia da linha e percebe que um ou mais processadores possuem uma cópia limpa em estado compartilhado, tal como descrito no item precedente.



(a) Linha na cache do processador requisitante



(b) Linha em cache que monitora o barramento

RH — Acerto em operação de leitura
RMS — Falha na leitura, linha compartilhada
RME — Falha na leitura, linha exclusiva
WH — Acerto em operação de escrita
WM — Falha em operação de escrita
SHR — Acerto de monitoramento na leitura
SHW — Acerto de monitoramento na escrita ou operação de leitura com intenção de modificação

(↓) Linha modificada copiada de volta na memória
(+) Transação de invalidação
(×) Leitura com intenção de modificação
(↑) Preenchimento de linha da cache

Figura 16.8 Diagrama de transição de estados do protocolo MESI.

Acerto na leitura

Quando ocorre um acerto na leitura (*read hit*) de uma linha correntemente na cache local, o processador simplesmente lê o item requerido. Não ocorre nenhuma mudança de estado da linha: o estado da linha permanece como modificada, compartilhada ou exclusiva.

Falha na escrita

Quando ocorre uma falha na escrita (*write miss*) na cache local, o processador inicia uma leitura na memória principal, para obter a linha da cache que contém o endereço da falha. Para isso, o processador envia um sinal por meio do barramento, que significa *leitura com intenção de modificação* (RWITM). Quando a linha é carregada na cache, ela imediatamente é marcada como modificada. Com relação às demais caches, duas possíveis situações precedem a carga da linha de dados.

Primeiro, alguma outra cache pode ter uma cópia modificada dessa linha (estado = modificada). Nesse caso, o processador alertado envia um sinal para o processador requisitante (que iniciou a transação), indicando que outro processador possui uma cópia modificada da linha. O processador requisitante então libera o barramento e aguarda. O outro processador obtém o controle do barramento, escreve a linha de cache modificada de volta na memória principal e altera o estado da linha na sua cache para inválida (porque o processador que iniciou a transação vai modificar essa linha). Subseqüentemente, o processador que iniciou a transação envia novamente um sinal RWITM no barramento e, então, lê a linha na memória principal, modifica a linha na cache e marca seu estado como modificada.

A outra situação possível é que nenhuma outra cache tenha uma cópia modificada da linha requisitada. Nesse caso, nenhum sinal é retornado e o processador requisitante procede a leitura e a modificação da linha. Enquanto isso, se uma ou mais caches possuíam uma cópia limpa da linha em estado compartilhado, cada uma dessas caches invalida sua cópia da linha; e se uma cache possui uma cópia limpa da linha em estado exclusivo, ela invalida sua cópia da linha.

Acerto na escrita

Quando ocorre um acerto na escrita (*write hit*) na cache local, o efeito depende do estado corrente da linha nessa cache:

- **Compartilhada**: antes de efetuar a atualização de dados na linha, o processador deve obter uso exclusivo dessa linha. Para isso, ele sinaliza sua intenção por meio do barramento. Cada processador que possua uma cópia compartilhada dessa linha em sua cache altera o estado da linha de compartilhada para inválida. O processador que iniciou a transação efetua a atualização e altera o estado da sua linha de compartilhada para modificada.
- **Exclusiva**: se o processador já detém controle exclusivo sobre a linha, ele simplesmente efetua a atualização e altera o estado da linha de exclusiva para modificada.
- **Modificada**: o processador já detém controle exclusivo da linha e ela já está marcada como modificada; portanto, ele simplesmente efetua a atualização.

Coerência entre caches L1 e L2

Os protocolos de coerência de cache descritos até agora baseiam-se na cooperação entre caches conectadas ao mesmo barramento ou em uma estrutura de interconexão SMP. Tipicamente, essas caches são caches L2 e o processador possui também uma cache L1, que não está diretamente conectada ao barramento e, portanto, não pode integrar o protocolo de monitoração. Portanto, é necessário algum esquema para manter a integridade de dados entre os dois níveis de cache, assim como entre todas as caches da configuração SMP.

A estratégia é estender o protocolo MESI (ou qualquer outro protocolo de coerência de cache) às caches L1. Assim, cada linha em uma cache L1 inclui bits para indicar o seu estado. Essencialmente, o objetivo é o seguinte: para qualquer linha que esteja presente na cache L2 e na cache L1 correspondente, o estado da linha na cache L1 deve ser igual ao seu estado na cache L2. Um mecanismo simples de garantir isso é adotar a política de escrita direta na cache L1; nesse caso, a escrita é feita na cache L2, e não na memória principal. A política de escrita direta na cache L1 faz com que qualquer modificação em uma linha de L1 seja reproduzida na cache L2 correspondente, tornando a modificação visível para as demais caches L2. O uso da política de escrita direta requer que o conteúdo da cache L1 seja um subconjunto do conteúdo da cache L2. Isso sugere, por sua vez, que a associatividade da cache L2 deve ser igual ou maior que a associatividade da cache L1. A política de escrita direta é usada no IBM S/390 SMP.

Se a cache L1 usar política de escrita na volta, a relação entre as duas caches é mais complexa. Existem diversas abordagens para manter coerência nesse caso. A abordagem usada no Pentium II é descrita detalhadamente em Shanley (1998).

16.4 CLUSTERS

Uma das áreas mais novas e promissoras de projeto de sistemas de computação é a de agregados ou aglomerados de computadores ou, simplesmente, *clusters*. A organização de clusters constitui uma alternativa para os multiprocessadores simétricos (SMP), como abordagem para prover alto desempenho e alta disponibilidade, e é particularmente atrativa para aplicações baseadas em servidores. Podemos definir um cluster como um grupo de computadores completos interconectados, trabalhando juntos, como um recurso de computação unificado, que cria a ilusão de constituir uma única máquina. O termo *computador completo* significa um sistema que pode operar por si próprio, independentemente do cluster; na literatura, cada computador componente do cluster é usualmente denominado um *nó*.

Brewer (1997) relaciona quatro benefícios que podem ser obtidos com a organização de clusters. Esses benefícios podem também ser vistos como objetivos ou requisitos de projeto desse tipo de organização:

- **Escalabilidade absoluta**: é possível criar clusters muito grandes, cuja capacidade de computação ultrapassa várias vezes a capacidade da maior máquina individual. Um cluster pode ser constituído de dezenas de máquinas, sendo cada uma um multiprocessador.
- **Escalabilidade incremental**: um cluster é configurado de maneira que seja possível adicionar novos sistemas, expandindo-o de forma incremental. Desse modo, um usuário pode começar com um sistema mais modesto e expandi-lo à medida que

crescem suas necessidades, sem ter de efetuar uma expansão mais radical, onde o sistema menor é completamente substituído por um sistema de maior porte.

- **Alta disponibilidade**: como cada nó de um cluster é um computador independente, uma falha em um nó não significa perda total de serviço. Em muitos produtos, a tolerância a falhas é tratada automaticamente pelo software.
- **Melhor relação custo/desempenho**: devido à facilidade de construir o sistema a partir de elementos ou nós básicos comercialmente disponíveis, é possível obter um cluster com poder de computação igual ou maior que uma única máquina de grande porte, com custo muito menor.

Configurações de clusters

Os clusters são classificados na literatura de várias formas diferentes. Talvez a classificação mais simples seja aquela baseada na forma como os computadores no cluster compartilham o acesso aos discos. A Figura 16.9a ilustra um cluster com dois nós, onde a única forma de interconexão é por meio de uma ligação de alta velocidade, que pode ser usada para troca de mensagens que coordenam a atividade do cluster. Essa ligação pode ser uma rede local (LAN), que é compartilhada com os outros computadores que não fazem parte do cluster, ou pode ser algum mecanismo de ligação dedicado. Nesse último caso, um ou mais computadores do cluster terão uma ligação com uma rede LAN ou WAN, de forma que existe uma conexão entre o cluster servidor e os sistemas cliente remotos. Note que, nessa figura, cada computador é representado como sendo um multiprocessador. Isso não é necessário, mas aumenta o desempenho e a disponibilidade.

Na classificação simples mostrada na Figura 16.9, a outra alternativa é um cluster em que os discos são compartilhados. Nesse caso, ainda existe, em geral, uma ligação para a troca de mensagens entre os nós. Além disso, existe um subsistema de disco que é ligado diretamente aos múltiplos computadores do cluster. Nessa figura, o subsistema de disco comum é um sistema RAID. É comum o uso de um sistema RAID ou de alguma outra tecnologia similar de discos redundantes, para que a alta disponibilidade existente em uma configuração com múltiplos computadores não seja comprometida por um sistema de disco compartilhado, que constitui um ponto central de falha.

Podemos ter uma visão mais clara sobre as opções de organização de clusters examinando as possíveis alternativas do ponto de vista funcional. Um artigo da Hewlett-Packard (HP, 1996) fornece uma classificação útil de clusters do ponto de vista funcional (Tabela 16.2), que é discutida a seguir.

Um método bastante comum e mais antigo, conhecido como **servidor secundário passivo** (*passive standby*), consiste simplesmente em deixar todo o processamento a cargo de um computador, enquanto outro computador permanece inativo, pronto para tomar o processamento em caso de falha no primeiro. Para coordenar as máquinas, o sistema ativo, ou primário, envia periodicamente uma mensagem de 'batida de coração' (*heartbeat*) para a máquina inativa, ou secundária. Se essas mensagens param de ser recebidas, o computador secundário considera que ocorreu uma falha no computador primário e se coloca em operação. Essa abordagem aumenta a disponibilidade do sistema, mas não melhora seu desempenho. Além disso, se a única informação trocada entre os dois sistemas é a mensagem de batida de coração e se os dois sistemas não compartilham discos comuns, então o computador secundário provê uma cópia funcional dos dados, mas não tem acesso aos bancos de dados gerenciados pelo sistema primário.

Tabela 16.2 Métodos de organização de clusters: benefícios e limitações

Método	Descrição	Benefícios	Limitações
Servidor secundário passivo	O servidor secundário apenas entra em operação em caso de falha no servidor primário.	Fácil de implementar.	Custo alto, uma vez que o servidor secundário não é usado para processamento de tarefas.
Servidor secundário ativo:	O servidor secundário também é usado para o processamento de tarefas.	Redução de custo, uma vez que servidores secundários são usados para processamento.	Maior complexidade.
• Servidores separados	Os servidores separados possuem seus próprios discos. Os dados são copiados continuamente do servidor primário para o servidor secundário.	Alta disponibilidade.	Alta sobrecarga de rede e de servidor, devido a operações de cópia.
• Servidores conectados a discos	Os servidores são ligados aos mesmos discos, mas cada servidor tem seu próprio conjunto de discos. Se um servidor falha, seus discos são tomados pelo outro servidor.	Redução da sobrecarga de rede e de servidor, devido à eliminação de operações de cópia.	Usualmente requer espelhamento de discos ou uso de tecnologia RAID para compensar os riscos de falha de discos.
• Servidores compartilhando discos	Múltiplos servidores compartilham acesso simultâneo aos discos.	Baixa sobrecarga de rede e de servidor. Risco reduzido de parada do sistema, devido a falha de disco.	Requer gerenciamento de bloqueio de acesso a discos por software. Normalmente é usado com espelhamento de discos ou tecnologia RAID.

Essa organização de servidor secundário passivo geralmente não é chamada cluster. O termo *cluster* é reservado para configurações com múltiplos computadores interconectados em que todos processam ativamente, provendo ao mundo externo a ilusão de constituir um sistema único. O termo **servidor secundário ativo** (*active secondary*) é freqüentemente usado para referenciar esse tipo de configuração. Três métodos de organização de clusters de computadores podem ser identificados: servidores separados, nenhum compartilhamento e discos compartilhados.

Na primeira abordagem, cada computador é um **servidor separado** (*separate server*), possuindo seus próprios discos e nenhum disco é compartilhado entre os diversos sistemas (Figura 16.9a). Esse arranjo provê alto desempenho, assim como alta disponibilidade. Nesse caso, é necessário algum tipo de software de gerenciamento ou de escalonamento, para atribuir aos servidores as requisições recebidas de clientes, de maneira que a carga de trabalho

seja balanceada entre os servidores e se obtenha uma alta taxa de utilização do sistema. É desejável também prover uma capacidade de tolerância a falhas, possibilitando que, em caso de falha de um computador em operação, outro computador do cluster possa concluir a execução da aplicação. Para isso, é necessário efetuar uma cópia de dados constante entre os sistemas, para que cada sistema tenha acesso aos dados correntes dos demais sistemas. A sobrecarga devida a essa cópia de dados assegura alta disponibilidade ao custo de uma diminuição no desempenho.

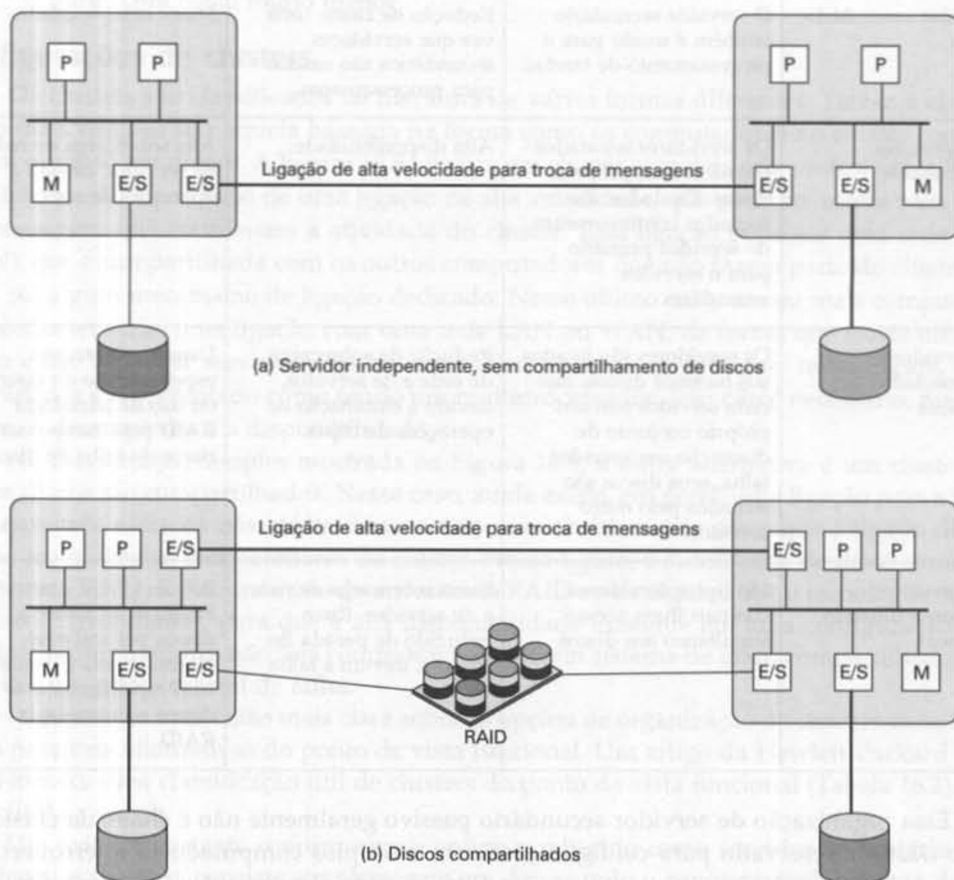


Figura 16.9 Configurações de clusters.

Para reduzir a sobrecarga de comunicação entre os nós, a maioria dos clusters consiste de servidores conectados a discos comuns (Figura 16.9b). Em uma variação dessa abordagem não há **nenhum compartilhamento** (*shared nothing*). Os discos comuns são particionados em volumes, e cada volume é propriedade de um único computador. Se esse computador falha, o cluster tem de ser reconfigurado, de modo que algum outro computador se torne o proprietário dos volumes pertencentes ao computador que falhou.

É também possível que múltiplos computadores compartilhem os mesmos discos simultaneamente (abordagem de **discos compartilhados** ou *shared disks*), de forma que cada com-

putador tem acesso a todos os volumes de todos os discos. Essa abordagem requer o uso de alguma facilidade de bloqueio de acesso, para assegurar que os dados sejam usados apenas por um computador de cada vez.

Questões de projeto de sistemas operacionais

Para explorar completamente as facilidades oferecidas por uma configuração de clusters, são necessárias algumas melhorias no sistema operacional, com relação aos sistemas operacionais voltados para sistemas com uma única máquina.

Gerenciamento de falhas

O gerenciamento de falhas depende do método de agregação usado (Tabela 16.2). Em geral, podem ser adotadas duas abordagens para lidar com falhas: clusters altamente disponíveis e clusters tolerantes a falha. Um cluster altamente disponível oferece alta probabilidade de que todos os recursos estejam em serviço. Se ocorre uma falha, tal como a queda de um sistema ou a perda de um volume de disco, as requisições em andamento são perdidas. Qualquer requisição perdida, se reiniciada, é atendida por outro computador do agregado. Entretanto, o sistema operacional do agregado não dá qualquer garantia sobre o estado das transações executadas parcialmente. Isso tem de ser tratado no nível da aplicação.

Um cluster tolerante a falha assegura que todos os recursos sempre estão disponíveis. Isso é obtido pelo uso de discos compartilhados redundantes e de mecanismos para manter cópias de segurança das transações ainda não confirmadas e para confirmar transações completadas.

A função de trocar aplicações e recursos de armazenamento de dados de um sistema em que ocorreu falha para um sistema alternativo do cluster é denominada **recuperação de falha** (*failover*). Uma função relacionada é o restabelecimento de aplicações e recursos de armazenamento de dados no sistema original, depois que ele é reparado; essa função é denominada **retorno à operação** (*failback*). O retorno de um sistema à operação pode ser automatizado, mas deve ser feito apenas se o defeito realmente foi reparado e tem pouca probabilidade de ocorrer novamente. Caso contrário, o retorno automático pode fazer com que os recursos de sistemas falhos fiquem passando de um computador para outro, resultando em problemas de desempenho e de recuperação do sistema.

Balanceamento de carga

Um cluster requer um balanceamento efetivo da carga de trabalho entre os diversos computadores disponíveis. Isso inclui a possibilidade de expandir o cluster de forma incremental. Quando um novo computador é adicionado ao cluster, o mecanismo de balanceamento de carga deve incluir automaticamente esse computador no procedimento de escalonamento de aplicações. Também devem existir mecanismos de *middleware* para reconhecer serviços que podem ser desempenhados nos diferentes membros do cluster e que podem ser migrados de um membro para outro.

Clusters *versus* SMP

Tanto clusters como multiprocessadores simétricos possuem uma configuração com múltiplos processadores para suportar aplicações com alta demanda de desempenho. As duas soluções são disponíveis comercialmente, embora os SMPs venham sendo usados há mais tempo.

A principal vantagem da abordagem SMP é que um SMP é mais fácil de ser configurado que um cluster. Um SMP é muito mais próximo do modelo original de um único processador, para o qual a maioria das aplicações foi escrita. A principal mudança requerida na passagem de um sistema uniprocessador para um SMP é na função de escalonamento. Outro benefício do SMP é requerer menor espaço físico e suprimento de energia que um cluster comparável. Um último benefício importante é que produtos SMP são bem estabelecidos e estáveis.

Entretanto, a longo prazo, as vantagens da abordagem de clusters tendem a fazer com que estes dominem o mercado de sistemas servidores de alto desempenho. Clusters são muito superiores aos SMPs em termos de escalabilidade absoluta e incremental. São também superiores em termos de disponibilidade, porque todos os componentes do sistema podem prontamente ser tornados altamente redundantes.

16.5 ACESSO NÃO-UNIFORME À MEMÓRIA (NUMA)

Em termos de produtos comerciais, as duas abordagens mais comuns para prover sistemas multiprocessadores para suportar aplicações são os clusters e os SMPs. Outra abordagem, conhecida como acesso não-uniforme à memória (NUMA), tem sido objeto de pesquisa há alguns anos e, recentemente, alguns produtos comerciais NUMA foram lançados.

Antes de prosseguir, devemos definir alguns termos freqüentemente encontrados na literatura referente a sistemas NUMA.

- **Acesso uniforme à memória (UMA)**: todos os processadores têm acesso a todas as partes da memória principal, via operações de carga e armazenamento. O tempo de acesso à memória por um processador é o mesmo para todas as regiões da memória. Os tempos de acesso experimentados por diferentes processadores também são iguais. A organização SMP discutida na Seção 16.2 é um sistema UMA.
- **Acesso não-uniforme à memória (NUMA)**: todos os processadores têm acesso a todas as partes da memória principal, via operações de carga e armazenamento. O tempo de acesso à memória por um processador difere conforme a região de memória que está sendo usada. Isso vale para todos os processadores; entretanto, as regiões de memória para as quais o acesso é mais lento ou mais rápido são diferentes para diferentes processadores.
- **NUMA com coerência de cache (CC-NUMA)**: um sistema NUMA no qual é mantida coerência de cache entre as memórias cache dos vários processadores.

Um sistema NUMA sem coerência de cache é mais ou menos equivalente a um cluster. Os produtos comerciais que têm recebido maior atenção recentemente são sistemas CC-NUMA, bastante diferentes dos clusters e SMPs. Usualmente, mas infelizmente nem sempre, tais sistemas são de fato relatados na literatura comercial como sistemas CC-NUMA. Esta seção aborda esse tipo de sistema.

Motivação

Em um sistema SMP, existe um limite prático para o número de processadores que podem ser usados. Um esquema de cache efetivo reduz o tráfego no barramento entre qualquer processador e a memória principal. À medida que aumenta o número de processadores, também aumenta o tráfego no barramento. Além disso, o barramento é usado para a troca de si-

nais do protocolo de coerência de cache, aumentando ainda mais o tráfego no barramento. A partir de determinado ponto, o barramento passa a constituir um gargalo de desempenho do sistema. A degradação de desempenho parece limitar o número de processadores de uma configuração SMP a algum valor entre 16 e 64 processadores. Por exemplo, o sistema SMP Power Challenge da Silicon Graphics é limitado a 64 processadores R10000 em um único sistema; além desse número, o desempenho degrada substancialmente.

A limitação do número de processadores em um SMP é uma das principais motivações para o desenvolvimento de sistemas de cluster. Entretanto, em um cluster, cada nó tem sua própria memória principal privativa; as aplicações não enxergam uma grande memória global. De fato, a coerência de dados é mantida por software e não por hardware. Essa granularidade da memória afeta o desempenho e, para obter maior desempenho, a aplicação deve ser adaptada para esse ambiente. Uma abordagem para obter um multiprocessamento em larga escala, mantendo o estilo SMP, é a abordagem NUMA. Por exemplo, o sistema NUMA Origin da Silicon Graphics é projetado para prover suporte a até 1024 processadores MIPS R10000 (Whitney et al, 1997) e o sistema NUMA-Q da Sequent é projetado para prover suporte a até 252 processadores Pentium II (Lovett e Clapp, 1996).

O objetivo de um sistema NUMA é manter, de forma transparente, uma visão de uma grande e única área de memória no sistema, permitindo, ao mesmo tempo, vários nós multiprocessadores, cada qual com seu próprio barramento ou outro sistema interno de interconexão.

Organização

A Figura 16.10 mostra uma organização CC-NUMA típica. Existem vários nós independentes, cada um dos quais sendo, de fato, uma organização SMP. Portanto, cada nó contém diversos processadores, cada qual com suas próprias caches L1 e L2, além da memória principal. Um nó constitui o bloco básico da organização CC-NUMA como um todo. Por exemplo, cada nó do sistema Origin da Silicon Graphics inclui dois processadores MIPS R10000; cada nó do sistema NUMA-Q da Sequent inclui quatro processadores Pentium II. Os nós são conectados por meio de algum sistema de comunicação, que pode ser um mecanismo de comutação, um anel ou algum tipo de rede.

Cada nó do sistema CC-NUMA inclui alguma memória principal. Entretanto, do ponto de vista dos processadores, existe uma única área de memória endereçável, com cada posição de memória tendo um endereço único em todo o sistema. Quando um processador inicia um acesso à memória, se a posição de memória requerida não está presente na cache do processador, a cache L2 inicia uma operação de busca. Se a linha requerida está na porção local da memória principal, ela é obtida por meio do barramento local. Se a linha está em uma porção remota da memória principal, então é automaticamente enviada uma requisição para buscar a linha por meio da rede de interconexão e entregue ao barramento local, que então a transfere para a cache requisitante conectada a ele. Toda essa atividade é feita de forma automática e transparente para o processador e para sua cache.

Nessa configuração, a coerência de cache é uma preocupação fundamental. Embora as implementações existentes difiram em alguns detalhes, podemos dizer, em termos gerais, que cada nó deve manter uma espécie de diretório, que indica a localização das várias porções da memória, assim como a informação de estado das caches. Para ver como esse esquema funciona, descrevemos um exemplo, extraído de Pfister (1998). Suponha que o processador 3 do nó 2 (P2-3) requisite acesso à posição de memória 798, que está localizada na memória do nó 1. Ocorrerá a seguinte seqüência de eventos:

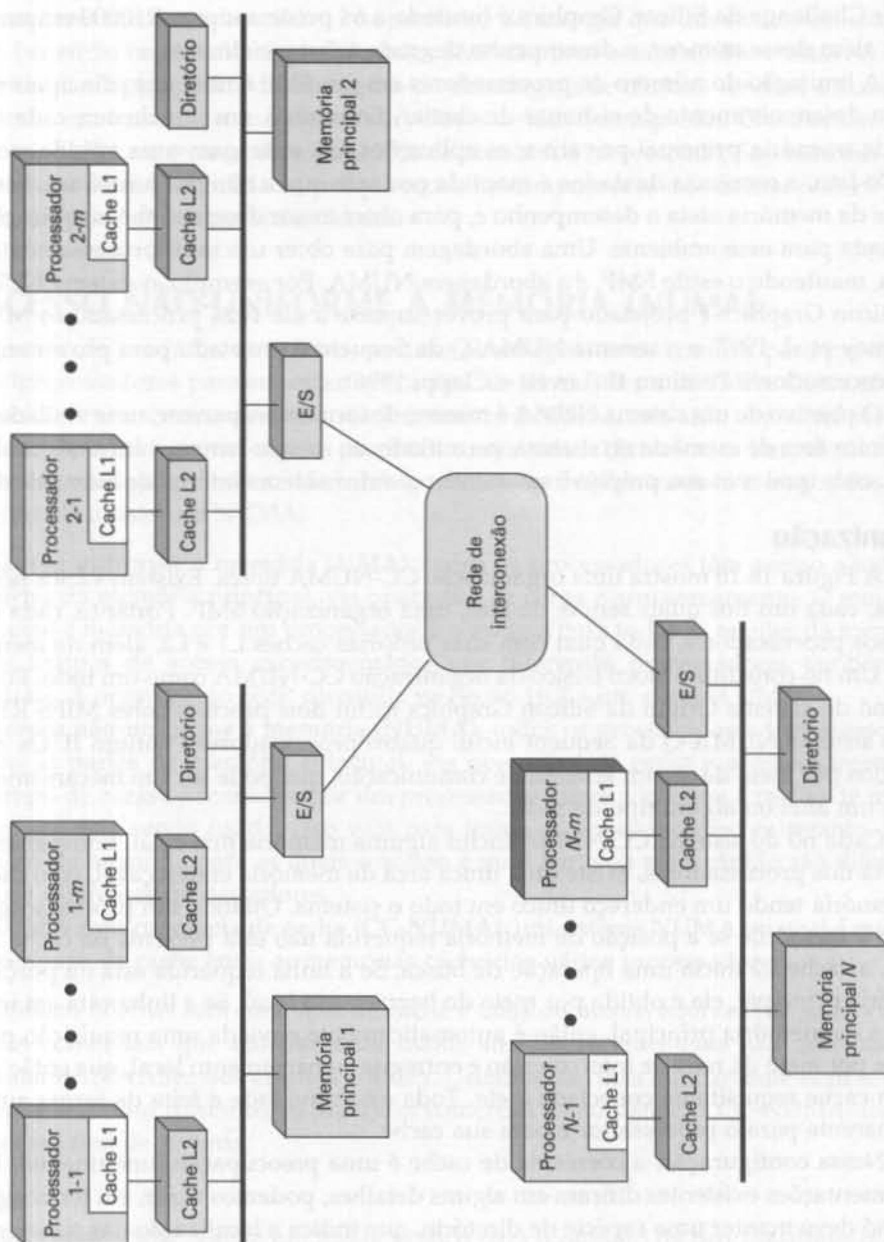


Figura 16.10 Organização CC-NUMA.

1. P2-3 envia uma requisição de acesso à posição 798, por meio do barramento de monitoração do nó 2.
2. O diretório do nó 2 detecta a requisição e verifica que essa posição está localizada no nó 1.
3. O diretório do nó 2 envia uma requisição ao nó 1, que é recebida pelo diretório do nó 1.
4. O diretório do nó 1, agindo como substituto de P2-3, requisita o conteúdo da posição de memória 798, como se ele fosse um processador.
5. A memória principal do nó 1 responde, colocando o dado requisitado no barramento.
6. O diretório do nó 1 obtém o dado do barramento.
7. O valor é transferido de volta para o diretório do nó 2.
8. O diretório do nó 2 coloca o dado no barramento do nó 2, agindo como substituto da memória principal que armazenava o dado originalmente.
9. O valor é obtido e colocado na cache de P2-3, e depois despachado para P2-3.

A seqüência precedente explica como um dado é lido de uma memória remota, usando mecanismos de hardware que tornam a transação transparente para o processador. No topo desses mecanismos, é requerido algum protocolo de coerência de cache. Os vários sistemas existentes diferem em como isso é feito exatamente. Faremos aqui apenas algumas observações a esse respeito. Primeiramente, como parte da seqüência precedente, o diretório do nó 1 mantém registro sobre algumas caches remotas que eventualmente tenham uma cópia da linha que contém a posição de memória 798. Deve haver, então, algum protocolo cooperativo para tratar operações de escrita. Por exemplo, se é feita uma modificação em uma cache, esse fato pode ser difundido para todas as demais caches. O diretório de cada nó que recebesse essa informação poderia, então, determinar se sua cache local contém uma cópia da linha referenciada e, em caso afirmativo, retirá-la da cache. Se a posição referenciada está localizada na memória principal de um nó que recebeu a notificação de difusão, então o diretório desse nó deve manter uma entrada indicando que essa linha da memória é inválida, permanecendo como tal até que ocorra uma escrita da linha de volta (*write-back*) na memória. Se outro processador (local ou remoto) requisitar a linha inválida, então, o diretório local deve forçar uma escrita da linha de volta na memória, antes de fornecer o dado requisitado.

Vantagens e desvantagens de sistemas NUMA

A principal vantagem de um sistema CC-NUMA é que ele pode disponibilizar desempenho efetivo em níveis de paralelismo mais altos que o fornecido por sistemas SMP, sem requerer mudanças substanciais no software. Com múltiplos nós NUMA, o tráfego no barramento em qualquer nó individual é limitado à demanda que esse barramento é capaz de tratar. Porém, se houver grande número de acessos a posições de memória localizadas em nós remotos, o desempenho cairá substancialmente. Entretanto, existem razões para acreditar que essa queda de desempenho pode ser evitada. Primeiramente, o uso de caches L1 e L2 é projetado para minimizar todos os acessos à memória principal, inclusive acessos remotos. Se a maioria dos programas tiver boa localidade temporal, o número de acessos remotos não deverá ser excessivo. Em segundo lugar, se os programas tiverem boa localidade espacial, e se é usada memória virtual, então os dados requeridos por uma aplicação deverão residir em um número limitado de páginas usadas freqüentemente, que podem ser inicialmente carregadas na memória local do processador que está executando a aplicação. Os projetistas da Sequent relatam que essa localidade espacial ocorre em aplicações representativas (Lovett, 1996). Finalmente, o esquema de memória virtual pode ser aprimorado, incluindo no sistema

operacional um mecanismo de migração de páginas, que seja capaz de mover uma página de memória virtual para um nó onde ela é usada freqüentemente. Os projetistas da Silicon Graphics reportam sucesso no uso dessa abordagem (Whitney et al, 1997).

Existem também algumas desvantagens na abordagem CC-NUMA. Duas delas em particular são discutidas com detalhes em Pfister (1998). A primeira é que um sistema CC-NUMA não se apresenta de forma transparente como um SMP; são requeridas modificações de software para migrar sistemas operacionais e aplicações de um sistema SMP para um sistema CC-NUMA. Tais modificações incluem alocação de páginas, como mencionado anteriormente, alocação de processos e balanceamento de carga pelo sistema operacional. Uma segunda preocupação é a disponibilidade. Essa é uma questão bastante complexa e depende da exata implementação do sistema CC-NUMA. Para maior aprofundamento você poderá consultar Pfister (1998).

16.6 COMPUTAÇÃO VETORIAL

Embora o desempenho de computadores de grande porte de propósito geral continue a melhorar sem parar, ainda há aplicações cujos requisitos estão além da capacidade de computação desses equipamentos mais modernos. Existe a necessidade de computadores capazes de resolver problemas matemáticos relativos a processos reais, tais como os que ocorrem em disciplinas como aerodinâmica, sismologia, meteorologia e física atômica, nuclear e de plasma (Wilson, 1984).

Tipicamente, esses problemas são caracterizados pela necessidade de alta precisão numérica e programas que efetuam repetidamente operações aritméticas de ponto flutuante em grandes vetores de números. A maioria desses problemas pertence à categoria conhecida como *simulação de campos contínuos*. Em essência, uma situação física pode ser descrita por uma superfície ou região em três dimensões (por exemplo, o fluxo de ar adjacente à superfície de um foguete). Essa superfície é aproximada por uma matriz de pontos. Um conjunto de equações diferenciais define o comportamento físico da superfície em cada ponto. As equações são representadas como uma matriz de coeficientes e valores, e a solução do sistema de equações envolve repetidas operações aritméticas sobre as matrizes de dados.

Para tratar desse tipo de problema, foram desenvolvidos os supercomputadores. Essas máquinas tipicamente são capazes de efetuar centenas de milhões de operações de ponto flutuante por segundo e custam entre 10 e 15 milhões de dólares. Em contraposição aos computadores de grande porte, que são projetados para multiprogramação e uso intensivo de E/S, os supercomputadores são otimizados para o tipo de computação numérica anteriormente descrita.

Os supercomputadores têm uso limitado e, devido ao seu alto custo, contam com um mercado limitado. Existe, comparativamente, um pequeno número de máquinas desse tipo em operação, a maioria delas em centros de pesquisa e agências governamentais com fins de desenvolvimento científico ou tecnológico. Assim como em outras áreas da tecnologia de computadores, existe uma constante demanda por maior desempenho de supercomputadores. Em algumas aplicações correntes em aerodinâmica e física nuclear, podem ser requeridas da ordem de 10^{13} operações aritméticas para resolver um problema simples, absorvendo mais de dois dias de tempo de computação em um supercomputador moderno. Por isso, a tecnologia e o desempenho de supercomputadores continuam a evoluir.

Outro tipo de sistema, também projetado visando à computação vetorial, é conhecido como *processador matricial*. Embora um supercomputador seja otimizado para computação vetorial, ele é também um computador de propósito geral, capaz de efetuar processamento escalar e tarefas de processamento de dados genéricas. Os processadores matriciais não incluem processamento escalar; eles são configurados como dispositivos periféricos de usuários de computadores de grande porte e minicomputadores para executar as partes vetorizadas de programas.

$$\begin{bmatrix} 1{,}5 \\ 7{,}1 \\ 6{,}9 \\ 100{,}5 \\ 0 \\ 59{,}7 \end{bmatrix} + \begin{bmatrix} 2{,}0 \\ 39{,}7 \\ 1000{,}003 \\ 11 \\ 21{,}1 \\ 19{,}7 \end{bmatrix} = \begin{bmatrix} 3{,}5 \\ 46{,}8 \\ 1006{,}903 \\ 111{,}5 \\ 21{,}1 \\ 79{,}4 \end{bmatrix}$$

$$A \quad + \quad B \quad = \quad C$$

Figura 16.11 Exemplo de adição de vetores.

Abordagens para processamento vetorial

A idéia básica do projeto de um supercomputador ou de um processador matricial é reconhecer que sua principal tarefa é efetuar operações aritméticas sobre vetores ou matrizes de números de ponto flutuante. Em um computador de propósito geral, isso pode requerer iteração sobre cada elemento do vetor ou matriz. Por exemplo, considere dois vetores (matrizes unidimensionais) de números, A e B. Desejamos somar esses dois vetores e armazenar o resultado em um vetor C. No exemplo da Figura 16.11, isso requer seis adições separadas. Como poderíamos aumentar a velocidade de computação? A resposta é introduzir alguma forma de paralelismo.

Diversas abordagens têm sido adotadas para obter paralelismo em computações vetoriais. Vamos ilustrá-las através de um exemplo. Considere a multiplicação de duas matrizes $C = A \times B$, onde A, B e C são matrizes $N \times N$. A fórmula para cada elemento de C é

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j}$$

onde A, B e C têm elementos $a_{i,j}$, $b_{i,j}$ e $c_{i,j}$, respectivamente. A Figura 16.12a mostra um programa FORTRAN para efetuar essa computação que pode ser executado em um processador escalar comum.

Uma abordagem para obter melhor desempenho pode ser denominada *processamento vetorial*. Ela inclui operações especiais para manipular um vetor de dados unidimensional. A Figura 16.12b é um programa FORTRAN com uma nova forma de instrução que possibilita especificar a computação vetorial. A notação (J = 1, N) indica que as operações sobre todos os índices J no intervalo especificado devem ser efetuadas como uma única operação. A forma como isso é feito é descrita a seguir.

```
          DO 100 I = 1, N
          DO 100 J = 1, N
          C(I, J) = 0.0
          DO 100 K = 1, N
          C(I, J) = C(I, J) + A(I, K) * B(K, J)
100       CONTINUE
```

(a) Processamento escalar

```
          DO 100 I = 1, N
          C(I, J) = 0.0 (J = 1, N)
          DO 100 K = 1, N
          C(I, J) = C(I, J) + A(I, K) * B(K, J) (J = 1, N)
100       CONTINUE
```

(b) Processamento paralelo

```
          DO 50 J = 1, (N–1)
          FORK 100
50        CONTINUE
          J = N
100       DO I = 1, N
          C(I, J) = 0.0
          DO 200 K = 1, N
          C(I, J) = C(I, J) + A(I, K) * B(K, J)
200       CONTINUE
          JOIN N
```

(c) Processamento vetorial

Figura 16.12 Multiplicação de matrizes (C = A × B).

O programa da Figura 16.12b indica que todos os elementos da i-ésima linha devem ser computados em paralelo. Cada elemento da linha é obtido como resultado de um somatório, e os somatórios (sobre K) são calculados de modo serial, e não em paralelo. Mesmo assim, são requeridas apenas N^2 multiplicações vetoriais, e não N^3 multiplicações escalares, como no algoritmo escalar.

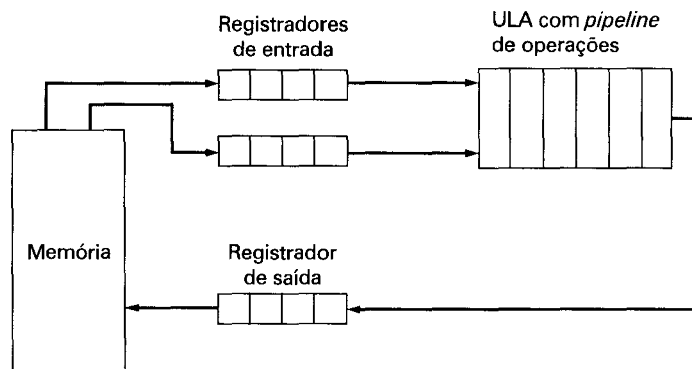
Outra abordagem, denominada *processamento paralelo*, é ilustrada na Figura 16.12c. Essa abordagem considera que existem *N* processadores independentes, que podem executar em paralelo. Para utilizar os processadores de modo efetivo, é necessário distribuir a computação entre eles, de alguma forma. Duas primitivas são usadas. A primitiva FORK *n* faz com que um processo independente seja iniciado a partir da instrução de endereço *n*. Enquanto isso, o processo original continua a executar a instrução imediatamente seguinte à instrução FORK. Cada execução de uma instrução FORK cria um novo processo. A instrução JOIN é, essencialmente, o inverso de FORK. Um comando JOIN N causa a combinação de *N* processos independentes em um único processo, que continua a execução a partir da instrução seguinte ao JOIN. O sistema operacional deve coordenar essa combinação, de forma que a execução apenas continue depois que todos os *N* processos alcançarem a instrução JOIN.

O programa da Figura 16.12c é escrito de modo que simule o processamento do programa de processamento vetorial. No programa de processamento paralelo, cada coluna da matriz C é computada por um processo separado. Assim, os elementos de uma mesma linha são computados em paralelo.

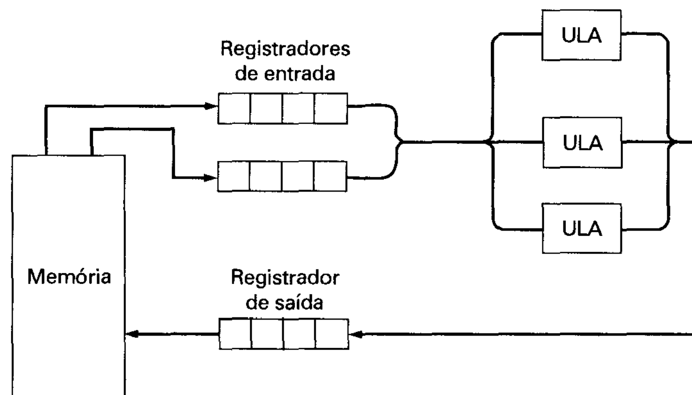
Na discussão precedente, descrevemos as abordagens para computação vetorial em termos lógicos ou arquiteturais. Vamos agora considerar os tipos de organização de processador que podem ser usados para implementar essas abordagens. Uma grande variedade de organizações tem sido experimentada. As principais categorias são:

- ULA com *pipeline* de operações
- ULAs paralelas
- Processadores paralelos

A Figura 16.13 ilustra as duas primeiras abordagens. O uso de *pipeline* foi discutido anteriormente no Capítulo 11. Esse conceito é estendido aqui para as operações executadas pela ULA. Como as operações de ponto flutuante são muito complexas, existe oportunidade para decompor uma operação de ponto flutuante em estágios, de maneira que diferentes estágios possam operar sobre diferentes conjuntos de dados, de forma concorrente. Isso é ilustrado na Figura 16.14a. A adição de números de ponto flutuante é dividida em quatro estágios (veja Figura 8.22): comparação, deslocamento, soma e normalização. Um vetor de números é submetido seqüencialmente ao primeiro estágio. À medida que prossegue o processamento, quatro diferentes conjuntos de números são operados de forma concorrente na *pipeline*.



(a) ULA com *pipeline* de operações



(b) ULAs paralelas

Figura 16.13 Abordagens para computação vetorial.

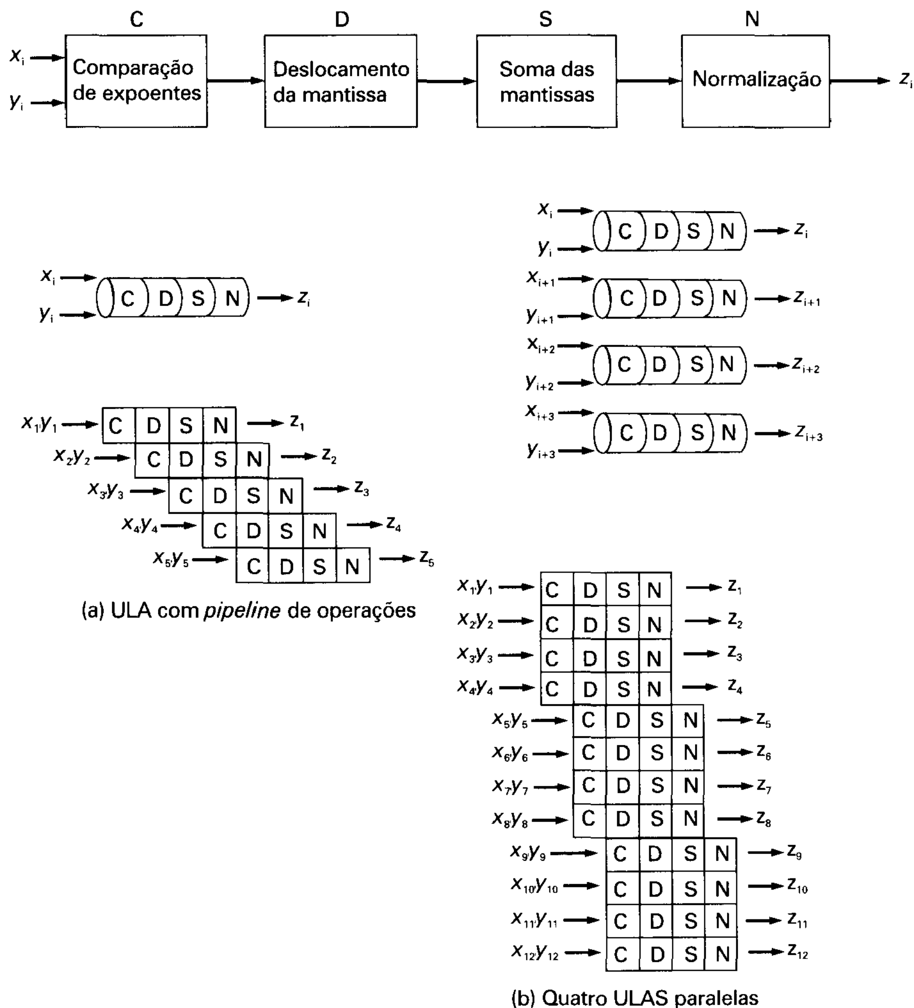


Figura 16.14 Processamento usando *pipeline*.

Deve ficar claro que essa organização é adequada para processamento vetorial. Para ver isso, considere a *pipeline* de instruções descrita no Capítulo 11. O processador efetua um ciclo repetitivo de busca e processamento de instruções. Na ausência de instruções de desvio, o processador busca instruções continuamente de posições consecutivas na memória. Conseqüentemente, a *pipeline* permanece cheia, resultando em economia de tempo. Analogamente, a *pipeline* da ULA apenas economiza tempo se for alimentada com uma seqüência de dados obtidos em posições de memória consecutivas. Uma única operação de ponto flutuante isolada não é executada mais rapidamente por uma *pipeline*. Apenas é obtido aumento de velocidade de processamento quando a ULA opera sobre um vetor de operandos. A unidade de controle envia dados para a ULA, até que todo o vetor seja processado.

A operação da *pipeline* pode ser melhorada ainda mais se os elementos do vetor são armazenados em registradores, e não são acessados diretamente da memória principal. Isso é

sugerido na Figura 16.13a. Os elementos do vetor são carregados como um bloco em um registrador vetorial, que consiste simplesmente em um grande banco de registradores idênticos. O resultado também é colocado em um registrador vetorial. Assim, a maioria das operações envolve apenas uso de registradores, e apenas as operações de carga e armazenamento e o início e o fim de uma operação vetorial requerem acesso à memória.

O mecanismo ilustrado na Figura 16.14 pode ser chamado de *pipeline dentro de uma operação*. Isto é, temos uma única operação aritmética (por exemplo, $C = A + B$), que é aplicada a vetores, e o uso de *pipeline* permite que múltiplos elementos do vetor sejam processados em paralelo. Esse mecanismo pode ser estendido com o uso de *pipeline entre operações*. Nesse caso, existe uma seqüência de operações que manipulam vetores, e a *pipeline* é usada para aumentar a velocidade de processamento dessa seqüência de instruções. Uma abordagem usada para isso, denominada *encadeamento* (*chaining*), é encontrada nos supercomputadores Cray. A regra básica é a seguinte: uma operação vetorial é iniciada tão logo o primeiro elemento do(s) operando(s) vetoral(is) esteja disponível e a unidade funcional requerida (por exemplo, adição, subtração, multiplicação, divisão) esteja livre. Essencialmente, o encadeamento faz com que resultados produzidos por uma unidade funcional sejam diretamente dirigidos como entrada para outra unidade funcional. Se forem usados registradores vetoriais, os resultados intermediários não precisam ser armazenados na memória, podendo ser usados antes mesmo que a operação vetorial que os produziu tenha sido completada.

Por exemplo, para computar $C = (s \times A) + B$, onde A, B e C são vetores e s é um escalar, o Cray pode executar três instruções simultaneamente. Os elementos vetoriais buscados na memória e carregados em registradores são imediatamente introduzidos na *pipeline* de multiplicação, os produtos são enviados para a *pipeline* de adição, e as somas são colocadas em um registrador vetorial tão logo são obtidas no somador:

1.	Carrega vetor	A → Registrador vetorial 1 (VR1)
2.	Carrega vetor	B → VR2
3.	Multiplica vetor	s × VR1 → VR3
4.	Soma vetores	VR3 + VR2 → VR4
5.	Armazena vetor	VR4 → C

As instruções 2 e 3 podem ser encadeadas (executadas simultaneamente na *pipeline*), pois envolvem diferentes registradores e posições de memória. A instrução 4 precisa dos resultados das instruções 2 e 3, mas também pode ser encadeada com elas. Tão logo os primeiros elementos dos registradores vetoriais estejam disponíveis, a operação da instrução 4 pode ser iniciada.

Outra forma de obter processamento vetorial é usar múltiplas ULAs em um mesmo processador, sob o controle de uma única unidade de controle. Nesse caso, a unidade de controle direciona dados para as ULAs, que funcionam em paralelo. É também possível usar *pipeline* em cada uma das ULAs paralelas. Isso é ilustrado na Figura 16.14b. O exemplo mostra quatro ULAs que operam em paralelo.

Assim como a organização em *pipeline*, a organização de ULAs paralelas é adequada para processamento vetorial. A unidade de controle direciona elementos de vetores para as ULAs, de forma circular, até que todos os elementos sejam processados. Esse tipo de organização é mais complexo que o de uma CPU com uma única ULA.

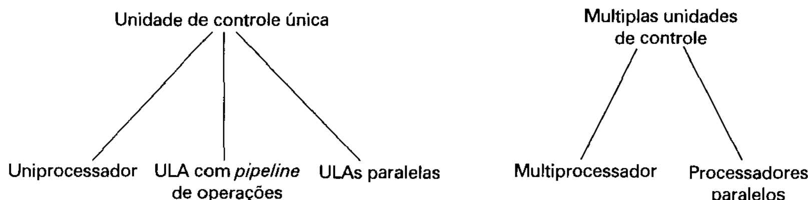


Figura 16.15 Uma taxonomia de organização de computadores.

Finalmente, também se pode obter processamento vetorial usando múltiplos processadores paralelos. Nesse caso, é necessário dividir a tarefa em múltiplos processos que possam ser executados em paralelo. Essa organização é efetiva apenas se existir efetiva coordenação dos processadores paralelos, pelo hardware e pelo software. Essa abordagem é ainda uma ativa área de pesquisa, embora já seja usada em alguns produtos (Gehringer, Abullarade e Gulyn, 1988).

A taxonomia da Seção 16.1 pode ser estendida de modo que reflita essas novas estruturas, conforme mostrado na Figura 16.15. Organizações de computadores podem ser distinguidas de acordo com a presença de uma ou mais unidades de controle. Múltiplas unidades de controle implicam múltiplos processadores. De acordo com nossa discussão anterior, se múltiplos processadores podem trabalhar de forma cooperativa sobre uma dada tarefa, eles são denominados *processadores paralelos*.

O leitor deve ser alertado sobre terminologias pouco adequadas usualmente encontradas na literatura. O termo *processadores vetoriais* (*vector processors*) freqüentemente é associado à organização de ULA com *pipeline* de operações, embora a organização de ULAs paralelas também seja projetada para processamento vetorial e, como discutimos previamente, uma organização de processadores paralelos possa também ser projetada para processamento vetorial. *Processamento matricial* (*array processing*) é algumas vezes usado para referir-se a ULAs paralelas, embora, novamente, qualquer das três organizações seja otimizada para processamento de vetores e matrizes. Para tornar as coisas ainda mais complicadas, o termo *processador matricial* (*array processor*) usualmente refere-se a um processador auxiliar acoplado a um processador de propósito geral e usado para efetuar computação vetorial. Um processador matricial pode usar a abordagem tanto de ULA com *pipeline* de operações como de ULAs paralelas.

No presente momento, a organização de ULA com *pipeline* de operações é a dominante no mercado. Sistemas baseados em *pipeline* são menos complexos que as demais abordagens. O projeto de sua unidade de controle e do seu sistema operacional está bem desenvolvido, possibilitando obter alocação eficiente de recursos e alto desempenho. O restante dessa seção é dedicado a um exame mais detalhado dessa abordagem, usando um exemplo específico.

Computação vetorial no IBM 3090

Um bom exemplo de organização da ULA com *pipeline* de operações para processamento vetorial é o mecanismo de processamento desenvolvido para a arquitetura IBM 370 e implementado na série 3090 (Padegs, Moore, Smith e Buchholz, 1988; Tucker, 1987). Esse mecanismo, denominado recurso vetorial (*vector facility*), é opcional e pode ser adicionado ao sistema básico, mas é bastante integrado a ele. É similar ao mecanismo de processamento vetorial encontrado em supercomputadores, tais como os da família Cray.

O mecanismo da IBM usa diversos registradores vetoriais. Cada registrador é, de fato, um banco de registradores escalares. Para computar a soma vetorial $C = A + B$, os vetores A e B são carregados em dois registradores vetoriais. Os dados desses registradores são passados para a ULA, o mais rápido possível, e os resultados são armazenados em um terceiro registrador vetorial. A sobreposição do processamento e da carga de dados nos registradores em bloco resulta em significativo aumento de desempenho com relação à operação de uma ULA normal.

Organização

A arquitetura vetorial da IBM, assim como as ULAs similares organizadas com *pipeline* de operações oferecem grande aumento de desempenho, em relação à execução repetida de operações escalares, pelas três formas a seguir:

- A estrutura fixa e predeterminada de vetores de dados possibilita que instruções internas ao laço de repetição sejam substituídas por operações de máquina internas (implementadas por hardware ou microcódigo), muito mais rápidas.
- Operações aritméticas e de acesso a dados, efetuadas sobre elementos sucessivos de um vetor, podem ser executadas concorrentemente, sobrepondo tais operações em uma *pipeline* ou efetuando operações sobre múltiplos elementos em paralelo.
- O uso de registradores vetoriais para armazenar resultados intermediários evita referências adicionais à memória.

A Figura 16.16 mostra a organização geral do mecanismo de processamento vetorial da IBM. Embora esse mecanismo seja visto como um recurso adicional separado fisicamente do processador, sua arquitetura constitui uma extensão da arquitetura do Sistema/370 e é compatível com ela. O mecanismo de processamento vetorial está integrado na arquitetura do Sistema/370 da seguinte forma:

- As instruções existentes no Sistema/370 são usadas para todas as operações escalares.
- Operações aritméticas sobre elementos individuais de um vetor produzem exatamente os mesmos resultados que instruções escalares correspondentes do Sistema/370. Por exemplo, uma decisão de projeto foi relacionada com a definição do resultado de uma operação de divisão de números de ponto flutuante. O resultado deveria ser exato, tal como em uma divisão de números de ponto flutuante escalares, ou deveria ser permitido um resultado aproximado, que permitiria uma implementação de maior velocidade, mas que, algumas vezes, introduzisse erro nos bits de mais baixa ordem? A decisão tomada foi a de manter total compatibilidade com o Sistema/370, mesmo que isso implicasse pequena degradação de desempenho.
- Instruções vetoriais podem ser interrompidas, sendo sua execução retomada do ponto em que ocorreu a interrupção, depois de tomada a ação apropriada, de forma compatível com o esquema de tratamento de interrupção de programa do Sistema/370.
- As exceções em operações aritméticas são as mesmas, ou são extensões, das exceções em instruções de aritmética escalar do Sistema/370; rotinas similares de recuperação de exceção podem ser usadas. Para isso, é empregado um índice de interrupção de vetor, que indica a posição do registrador vetorial que é afetada pela exceção (por exemplo, *overflow*). Portanto, quando a execução de uma instrução vetorial é retomada, a posição adequada do registrador vetorial é usada.

- Dados vetoriais residem na memória virtual, com as faltas de página tratadas da forma padrão.

Esse nível de integração oferece uma série de benefícios. Sistemas operacionais existentes podem oferecer suporte ao mecanismo de processamento vetorial, requerendo pequenas extensões. Programas de aplicação, compiladores e outros softwares existentes podem ser executados sem requerer qualquer modificação. Softwares que podem obter vantagem do mecanismo de processamento vetorial podem ser modificados, se desejado.

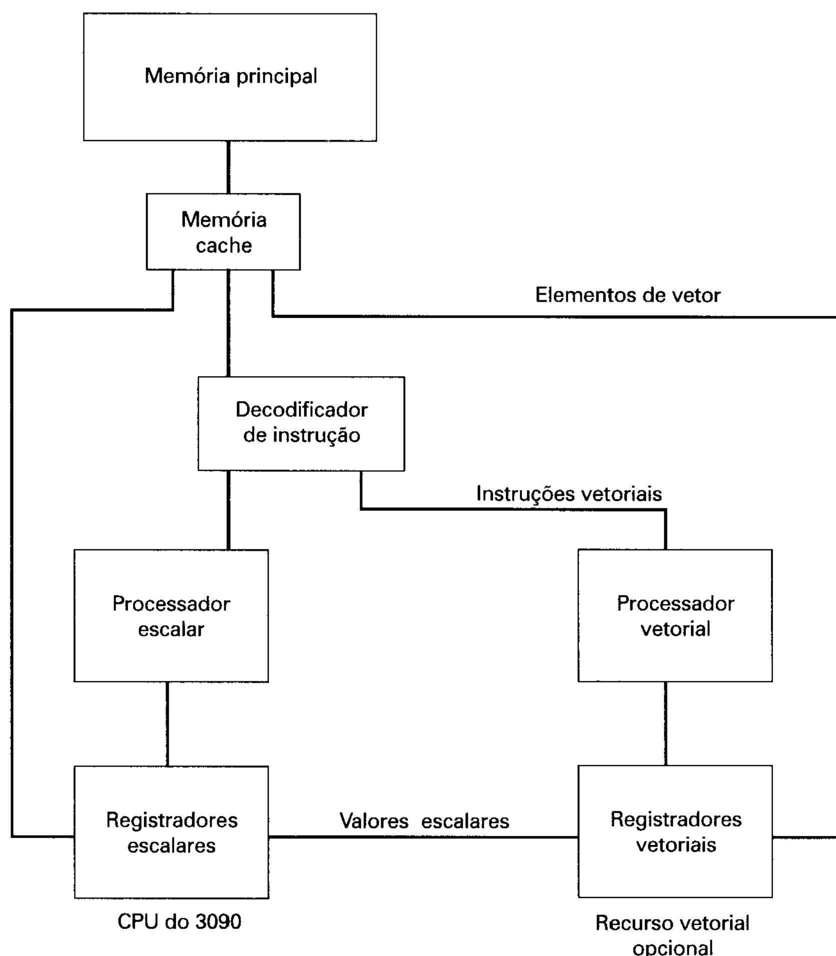


Figura 16.16 Mecanismo de processamento vetorial do IBM 3090.

Registradores

Uma característica-chave do projeto de um mecanismo de processamento vetorial é a de os operandos serem armazenados em registradores ou na memória. A organização da IBM é denominada *registrador para registrador*, uma vez que operandos vetoriais, tanto de entrada como de saída, podem ser mantidos em registradores vetoriais. Essa abordagem também é

usada no supercomputador Cray. Uma abordagem alternativa, usada nas máquinas Control Data, é obter os operandos diretamente da memória. A principal desvantagem do uso de registradores é que o programador ou compilador tem de gerenciar o seu uso. Por exemplo, suponha que o comprimento de um registrador vetorial é *K* e que o comprimento do vetor a ser processado é $N > K$. Nesse caso, deve ser efetuado um laço de repetição em que a operação vetorial é feita sobre *K* elementos de cada vez, e o laço é repetido N/K vezes. A principal vantagem da abordagem de registradores vetoriais é que a operação é isolada da memória principal, mais lenta, sendo feita primordialmente com registradores.

ROTINA FORTRAN:

```
      DO 100 J = 1, 50
      CR (J) = AR (J) *BR (J) - AI (J) *BI (J)
100   CI (J) = AR (J) *BI (J) - AI (J) *BR (J)
```

Operação	Ciclos
AR (J) *BR (J) → T1 (J)	3
AI (J) *BI (J) → T2 (J)	3
T1 (J) -T2 (J) → CR (J)	3
AR (J) *BI (J) → T3 (J)	3
AI (J) *BR (J) → T4 (J)	3
T3 (J) +T4 (J) → CI (J)	3
TOTAL	18

(a) Memória para memória

Operação	Ciclos
AR (J) → V1 (J)	1
BR (J) → V2 (J)	1
V1 (J) *V2 (J) → V3 (J)	1
AI (J) → V4 (J)	1
B1 (J) → V5 (J)	1
V4 (J) *V5 (J) → V6 (J)	1
V3 (J) -V6 (J) → V7 (J)	1
V7 (J) → CR (J)	1
V1 (J) *V5 (J) → V8 (J)	1
V4 (J) *V2 (J) → V9 (J)	1
V8 (J) +V9 (J) → V0 (J)	1
V0 (J) → C1 (J)	1
TOTAL	12

(b) Registrador para registrador

Operação	Ciclos
AR (J) → V1 (J)	1
V1 (J)*BR (J) → V2 (J)	1
AI (J) → V3 (J)	1
V3 (J)*BI (J) → V4 (J)	1
V2 (J)–V4 (J) → V5 (J)	1
V5 (J) → CR (J)	1
V1 (J)*BI (J) → V6 (J)	1
V3 (J)*BR (J) → V7 (J)	1
V6 (J)+V7 (J) → V8 (J)	1
V8 (J) → CI (J)	1
TOTAL	10

(c) Memória para registrador

Operação	Ciclos
AR (J) → V1 (J)	1
V1 (J)*BR (J) → V2 (J)	1
AI (J) → V3 (J)	1
V2 (J)–V3 (J) *B1 (J) → V2 (J)	1
V2 (J) → CR (J)	1
V1 (J)*BI (J) → V4 (J)	1
V4 (J)+V3(J) *BR (J) → C4 (J)	1
V4 (J) CI (J)	1
TOTAL	8

(d) Instruções compostas

Vi = registradores vetoriais
AR, BR, AI, BI = operandos na memória
Ti = posições de armazenamento temporário na memória

Figura 16.17 Programas alternativos para cálculos sobre vetores.

O aumento de velocidade que pode ser alcançado usando registradores é demonstrado na Figura 16.17 (Padegs, Moore, Smith e Buchholz, 1988). A rotina FORTRAN multiplica o vetor A pelo vetor B, produzindo o vetor C, onde cada vetor tem uma parte real (AR, BR, CR) e uma parte imaginária (AI, BI, CI). O 3090 pode efetuar um acesso à memória (para leitura

ou escrita) por ciclo de processador ou ciclo de relógio, tem registradores que podem sustentar dois acessos de leitura e um acesso de escrita por ciclo e produz um resultado por ciclo na sua unidade aritmética. Suponha que são usadas instruções que podem especificar dois operandos fonte e um resultado.[4] A parte (a) da figura mostra que, com instruções de memória para memória, cada iteração da computação requer um total de 18 ciclos. Com uma arquitetura registrador para registrador pura (parte (b)), esse tempo é reduzido para 12 ciclos. É claro que, com operações registrador para registrador, os vetores devem ser carregados em registradores vetoriais antes da computação e armazenados na memória depois. Para vetores muito grandes, essa penalidade fixa é relativamente pequena. A Figura 16.17c mostra que a habilidade de especificar em uma instrução tanto operandos na memória quanto em registradores reduz ainda mais o tempo, para 10 ciclos por iteração. Esse último tipo de instrução é incluído na arquitetura vetorial.[5]

A Figura 16.18 ilustra os registradores que fazem parte do mecanismo de processamento vetorial do IBM 3090. Existem 16 registradores vetoriais de 32 bits. Os registradores vetoriais também podem ser acoplados de modo que formatem 8 registradores vetoriais de 64 bits. Qualquer registrador pode armazenar um valor inteiro ou de ponto flutuante. Portanto, os registradores vetoriais podem ser usados para valores inteiros de 32 bits e de 64 bits, assim como para valores de ponto flutuante de 32 bits e de 64 bits.

A arquitetura especifica que cada registrador contém de 8 a 512 elementos escalares. A escolha do tamanho real envolve algumas decisões de projeto. O tempo para executar uma operação vetorial consiste, essencialmente, da sobrecarga inicial de preenchimento da *pipeline* e dos registradores, mais um ciclo por elemento do vetor. Portanto, o uso de um grande número de elementos de registrador reduz o tempo relativo de início da computação. Entretanto, essa eficiência deve ser balanceada em relação ao tempo adicional requerido para salvar e restaurar registradores vetoriais, na ocorrência de troca de processos, assim como em relação a limites práticos de custo e espaço. Essas considerações levaram ao uso, na implementação corrente do 3090, de 128 elementos por registrador.

Três registradores adicionais são necessários. O *registrador de máscara vetorial* contém bits de máscara que podem ser usados para selecionar elementos de um registrador vetorial que serão processados por uma determinada operação. O *registrador de estado vetorial* contém campos de controle, tais como o contador de vetor, que determina quantos elementos do registrador vetorial serão processados. O registrador *contador de atividade vetorial* mantém informação sobre o tempo gasto na execução de instruções vetoriais.

Instruções compostas

Como foi discutido anteriormente, a execução de instruções pode ser sobreposta usando encadeamento, para aumentar o desempenho. Os projetistas do mecanismo de processamento vetorial da IBM escolheram não incluir essa facilidade, por diversas razões. A arquitetura do Sistema/370 teria de ser estendida para lidar com interrupções complexas (incluindo seu efeito no gerenciamento de memória virtual) e seriam requeridas mudanças correspondentes no

4 Nas arquiteturas IBM /370 e /390, apenas as instruções de três operandos (instruções de registrador e memória, RS) especificam dois operandos em registradores e um na memória. Na parte (a) do exemplo, supomos a existência de instruções com três operandos na memória principal, apenas para propósito de comparação. De fato, tal formato de instrução poderia ter sido escolhido para a arquitetura vetorial.

5 Instruções compostas (*compound instructions*), discutidas em seguida, apresentam uma redução adicional.

software. Uma questão mais importante foi o custo de incluir controles adicionais e caminhos de acesso a registradores no mecanismo de processamento vetorial, para prover um encadeamento generalizado.

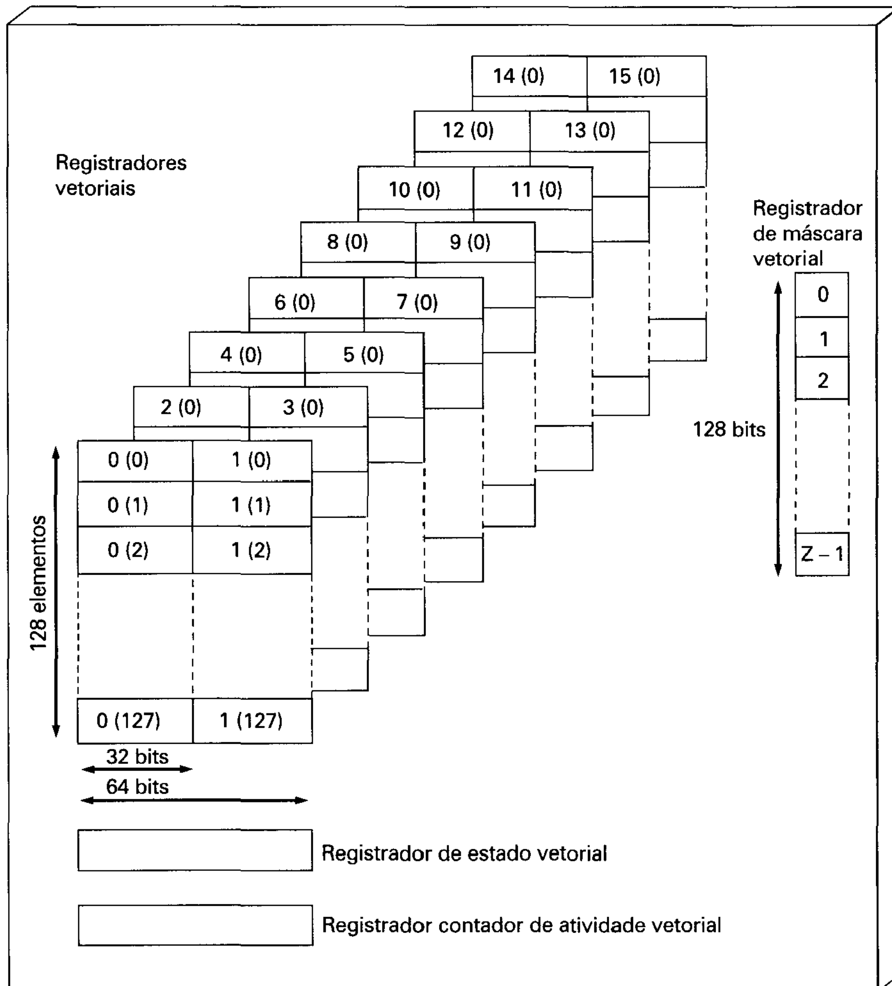


Figura 16.18 Registradores do mecanismo de processamento vetorial do IBM 3090.

Em vez disso, foram disponibilizadas três operações que combinam em uma instrução (um código de operação) as seqüências de computação de vetores mais comuns, ou seja, multiplicação seguida de adição, de subtração ou de somatório. A instrução MULTIPLY-AND-ADD de memória para registrador, por exemplo, busca um vetor na memória, multiplica esse vetor por um vetor armazenado em registrador e soma o produto a um terceiro vetor armazenado em registrador. Com o uso de instruções compostas MULTIPLY-AND-ADD e MULTIPLY-AND-SUBTRACT no exemplo da Figura 16.17, o tempo total para cada iteração é reduzido de 10 para 8 ciclos.

Ao contrário do encadeamento, as instruções compostas não requerem uso de registradores adicionais para armazenamento temporário de resultados intermediários e requerem um acesso a registrador a menos. Por exemplo, considere o seguinte encadeamento:

$$A \rightarrow VR1$$

$$VR1 + VR2 \rightarrow VR1$$

Nesse caso, são requeridas duas escritas no registrador VR1. Na arquitetura IBM, existe uma instrução de soma de memória para registrador. Com essa instrução, apenas o resultado da soma é armazenado em VR1. Instruções compostas também evitam a necessidade de refletir a execução concorrente de diversas instruções na descrição do estado da máquina, o que simplifica o tratamento de interrupções, assim como as operações de salvamento e restauração de contextos de execução efetuadas pelo sistema operacional.

O conjunto de instruções

A Tabela 16.3 resume as operações aritméticas e lógicas definidas para a arquitetura vetorial. Além dessas, existem instruções de carga da memória para registrador e instruções de armazenamento de registrador para a memória. Note que muitas das instruções usam um formato com três operandos. Além disso, muitas instruções têm diversas variantes, dependendo da localização dos operandos. Um operando fonte pode estar em um registrador vetorial (V), na memória (M) ou em um registrador escalar (E). O operando de destino é sempre um registrador vetorial, exceto em uma comparação, cujo resultado é armazenado no registrador de máscara vetorial. Com todas essas variantes, o número total de códigos de operação (instruções distintas) é 171. Esse número é bastante grande, entretanto, sua implementação não é tão cara como se poderia imaginar. Uma vez que a máquina disponha de unidades aritméticas e caminhos de dados para buscar operandos da memória, de registradores escalares e de registradores vetoriais para alimentar a *pipeline* de instruções vetoriais, a maior parte do custo de hardware já terá sido despendida. Com pequeno custo adicional, a arquitetura pode oferecer um rico conjunto de variantes de instruções que usam esses registradores e *pipelines*.

A maioria das instruções da Tabela 16.3 é auto-explicativa. As duas instruções de somatório precisam de maior explicação. Uma instrução de *acumulação* soma todos os elementos de um vetor (ACCUMULATE) ou elementos do produto de dois vetores (MULTIPLY- AND-ACCUMULATE). Essas instruções apresentam um problema de projeto interessante. Gostaríamos que essa operação fosse efetuada o mais rápido possível, aproveitando ao máximo a *pipeline* da ULA. A dificuldade é que a soma de dois números colocados na *pipeline* apenas fica disponível vários estágios adiante. Assim, o terceiro elemento do vetor não pode ser adicionado à soma dos dois primeiros antes que esses dois elementos tenham atravessado toda a *pipeline*. Para resolver esse problema, os elementos do vetor são somados de forma que produzam quatro somas parciais. Em particular, a soma dos elementos 0, 4, 8, 12, ..., 124 produz a soma parcial 0; os elementos 1, 5, 9, 13, ..., 125 produzem a soma parcial 1; os elementos 2, 6, 10, 14, ..., 126, a soma parcial 2; e os elementos 3, 7, 11, 15, ..., 127, a soma parcial 3. Cada uma dessas somas parciais pode prosseguir através da *pipeline* na velocidade máxima, pois o atraso na *pipeline* é de aproximadamente quatro ciclos. Um registrador vetorial separado é usado para armazenar as somas parciais. Quando todos os elementos do vetor original tiverem sido processados, as quatro somas parciais são adicionadas para produzir o resultado final. O desempenho dessa segunda fase não é crítico porque estão envolvidos apenas quatro elementos de vetor.

Tabela 16.3 Processamento vetorial do IBM 3090: instruções aritméticas e lógicas

Operação	Tipos de dados: Ponto flutuante: Longo	Tipos de dados: Ponto flutuante: Curto	Tipos de dados: Binário ou lógico	Localizações de operandos			
Soma	PFL	PFC	IB	V + V → V	V + M → V	E + V → V	E + M → V
Subtração	PFL	PFC	IB	V − V → V	V − M → V	E − V → V	E − M → V
Multiplicação	PFL	PFC	IB	V × V → V	V × V → V	E × V → V	E × M → V
Divisão	PFL	PFC	—	V / V → V	V / M → V	E / V → V	E / M → V
Comparação	PFL	PFC	IB	V · V → V	V · M → V	E · V → V	E · M → V
Multiplicação e soma	PFL	PFC	—		V + V × M → V	V + E × V → V	V + E × M → V
Multiplicação e subtração	PFL	PFC	—		V − V × M → V	V − E × V → V	V − E × M → V
Multiplicação e acumulação	PFL	PFC	—	P + · V → V	P + · M → V		
Complemento	PFL	PFC	IB	−V → V			
Absoluto Positivo	PFL	PFC	IB	$\vert V \vert \to V$			
Absoluto Negativo	PFL	PFC	IB	$-\vert V \vert \to V$			
Máximo	PFL	PFC	—			E · V → E	
Absoluto Máximo	PFL	PFC	—			E · V → E	
Mínimo	PFL	PFC	—			E · V → E	
Deslocamento lógico à esquerda	—	—	LO	· V → V			
Deslocamento lógica à direita	—	—	LO	· V → V			
AND	—	—	LO	V & V → V	V & M Ä V	E & V → V	E & M → V
OR	—	—	LO	$V \vert V \to V$	$V \vert M \to V$	$E \vert V \to V$	$E \vert M \to V$
XOR	—	—	LO	V ⊕ V → V	V ⊕ M → V	E ⊕ V → V	E ⊕ M → V

Explanação:

Tipos de Dados
PFL = Ponto Flutuante Longo
PFC = Ponto Flutuante Curto
IB = Inteiro Binário
LO = Lógico

Localizações de Operandos
V = Registrador Vetorial
M = Memória
E = Escalar (registrador de propósito geral ou de ponto flutuante)
P = Soma parcial em registrador vetorial
· = Operação especial

16.7 LEITURAS RECOMENDADAS

Uma visão geral dos princípios de multiprocessadores é apresentada em Catanzaro (1994), que também examina mais detalhadamente SMPs baseados na arquitetura SPARC. Os SMPs são também abordados com algum detalhe em Stone (1993) e Hwang (1993). Uma leitura essencial para qualquer um interessado em clusters é Pfister (1998); o livro aborda questões do projeto do hardware e do software e compara a organização de clusters com sistemas SMP e NUMA; ele contém também uma sólida descrição técnica sobre projeto de sistemas SMP e NUMA.

Uma excelente revisão de questões relativas à coerência de cache em multiprocessadores é apresentada em Lilja (1993). Vários artigos fundamentais sobre esse assunto estão coletados em Tomasevic e Milutinovic (1993).

Uma boa discussão sobre computação vetorial pode ser encontrada em Stone (1993) e Hwang (1993).

16.8 EXERCÍCIOS

16.1 Seja α a porcentagem de código de um programa que pode ser executada simultaneamente por n processadores em um sistema de computador. Suponha que o código restante deve ser executado seqüencialmente por um único processador. Cada processador tem taxa de execução de x MIPS.

a. Obtenha uma expressão para a taxa MIPS efetiva, quando o sistema é usado exclusivamente para a execução desse programa, em termos de n, α e x.

b. Se $n = 16$ e $x = 4$ MIPS, determine o valor de α que resulta em desempenho do sistema igual a 40 MIPS.

16.2 Um multiprocessador com oito processadores tem conectadas 20 unidades de fita. Um grande número de tarefas é submetido ao sistema e cada uma requer, no máximo, quatro unidades de fita para completar sua execução. Suponha que cada tarefa inicia usando apenas três unidades de fita por um longo período, antes de requerer a quarta unidade de fita, por um período curto, ao final de sua execução. Suponha também que existe uma entrada infinita de tarefas desse tipo.

a. Suponha que o escalonador do sistema operacional não inicia uma tarefa antes que existam quatro unidades de fita disponíveis. Quando uma tarefa é iniciada, as quatro unidades de fita são imediatamente alocadas a ela e não são liberadas até que a tarefa seja completada. Qual é o número máximo de tarefas que podem ser executadas ao mesmo tempo? Qual é o maior e o menor número de unidades de fita que ficam inativas como resultado dessa política de alocação?

b. Sugira uma política alternativa para que melhore a utilização das unidades de fita e, ao mesmo tempo, evite a ocorrência de bloqueio perpétuo do sistema. Qual é o número máximo de tarefas que podem ser executadas ao mesmo tempo? Quais são os limites para o número de unidades de fita ociosas?

16.3 Você pode vislumbrar algum problema com a abordagem de escrita única (*write-once*) na cache em um multiprocessador com arquitetura baseada em barramento? Em caso afirmativo, sugira uma solução.

16.4 Considere a situação em que dois processadores em uma configuração SMP requerem acesso, várias vezes, à mesma linha de dados da memória. Os dois processadores pos-

suem uma cache e usam o protocolo MESI. Inicialmente, as duas caches possuem uma cópia inválida da linha. A Figura 16.19 mostra as conseqüências de uma leitura da linha x pelo processador P1. Se esse é o início de uma seqüência de acessos, desenhe as figuras subseqüentes, para a seguinte seqüência:

1. P2 lê x.
2. P1 escreve em x (para maior clareza, identifique a linha na cache de P1 como x').
3. P1 escreve em x (para maior clareza, identifique a linha na cache de P1 como x'').
4. P2 lê x.

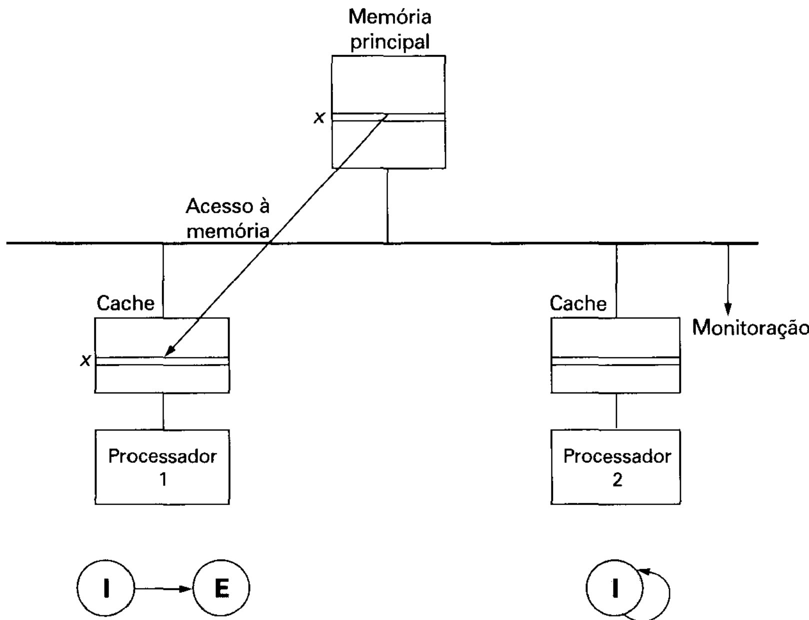


Figura 16.19 Exemplo de protocolo MESI: processador 1 lê a linha x.

16.5 A Figura 16.20 mostra dois diagramas de transição de estados de possíveis protocolos de coerência de cache. Estude, deduza e explique cada um desses protocolos e compare cada um com o protocolo MESI.

16.6 Considere um SMP com caches L1 e L2, usando o protocolo MESI. Como explicado na Seção 16.3, um dos quatro estados é associado a cada linha da cache L2. Os quatro estados são necessários para linhas da cache L1? Se sim, por quê? Se não, explique qual ou quais estados podem ser eliminados.

16.7 A Tabela 16.1 mostra o desempenho do arranjo com três níveis de cache do IBM S/390. O propósito desse problema é determinar se a inclusão do terceiro nível de cache vale a pena. Determine a penalidade de acesso à cache (número médio de ciclos da unidade de processamento), para um sistema que disponha apenas da cache L1, e normalize esse valor para 1.0. Determine, então, a penalidade de acesso quando são usadas as caches L1 e L2 e a penalidade de acesso quando as três caches são usadas. Note o percentual de melhoria em cada caso e dê sua opinião sobre a importância da cache L3.

16.8 O seguinte segmento de código deve ser executado 64 vezes para avaliar a expressão aritmética vetorial: $D(I) = A(I) + B(I) \times C(I)$, para $0 \leq I \leq 63$.

```
Load R1, B(I)       /R1 ← Memória (α + I)/
Load R2, C(I)       /R2 ← Memória (β + I)/
Multiply R1, R2     /R1 ← (R1) × (R2)/
Load R3, A(I)       /R3 ← Memória (γ + I)/
Add R3, R1          /R3 ← (R3) + (R1)/
Store D(I), R3      /Memória(θ + I) ← (R3)/
```

onde R1, R2 e R3 são registradores do processador e α, β, γ e θ são os endereços iniciais de memória dos vetores *B*(*I*), *C*(*I*), *A*(*I*) e *D*(*I*), respectivamente. Suponha que cada operação de carga ou armazenamento gaste 4 ciclos, a soma gaste 2 ciclos e a multiplicação gaste 8 ciclos, em um sistema uniprocessador ou em um processador de uma máquina SIMD.

a. Calcule o número total de ciclos de processador necessários para executar seqüencialmente esse segmento de código 64 vezes em um computador SISD uniprocessador, ignorando todos os outros atrasos.

b. Considere o uso de um computador SIMD com 64 elementos de processamento, que podem executar operações de seis instruções vetoriais sincronizadas, sobre vetores de dados de 64 componentes, todas elas controladas pelo mesmo relógio. Calcule o tempo total de execução na máquina SIMD, ignorando atrasos devidos à difusão das instruções e outros atrasos.

c. Qual é o aumento de velocidade de processamento obtido com computador SIMD em relação ao computador SISD?

16.9 Escreva uma versão vetorizada do seguinte programa:

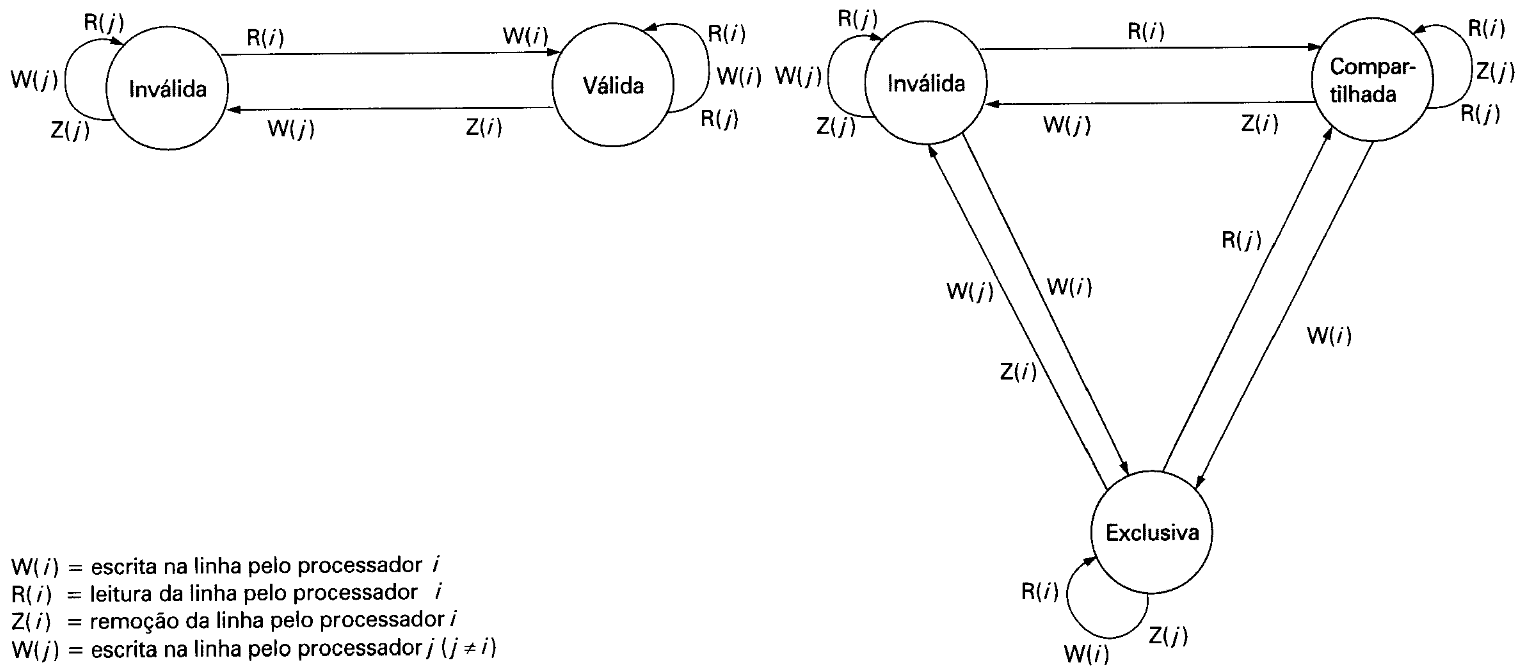
```
        DO 20 I = 1, N
        B(I, J) = 0
        DO 10 J = 1, M
        A(I) = A(I) + B(I, J) × C(I, J)
10      CONTINUE
        D(I) = E(I) + A(I)
20      CONTINUE
```

16.10 Um computador uniprocessador pode operar em modo escalar ou vetorial, onde a computação é efetuada nove vezes mais rápido no modo vetorial. Um programa de benchmark usado para comparação de desempenho foi executado em um tempo T nesse computador. Desse tempo, 25% foi no modo vetorial e o restante no modo escalar.

a. Calcule o aumento efetivo de velocidade de processamento sob as condições mencionadas, em comparação com a situação em que o modo vetorial não é usado. Calcule também o percentual de código que foi vetorizado (compilado para usar o modo vetorial), α, no programa precedente.

b. Suponha que dobramos a razão entre as velocidades nos modos vetorial e escalar, por meio de melhorias no hardware. Calcule o aumento efetivo de velocidade obtido.

c. Suponha que o mesmo aumento de velocidade obtido em (b) é obtido por meio de melhoria no compilador, e não no hardware. Qual seria a nova taxa de vetorização α que o compilador deveria produzir para o mesmo programa de benchmark?



W(i) = escrita na linha pelo processador i
R(i) = leitura da linha pelo processador i
Z(i) = remoção da linha pelo processador i
W(j) = escrita na linha pelo processador j ($j \neq i$)
R(j) = leitura da linha pelo processador j ($j \neq i$)
Z(j) = remoção da linha pelo processador j ($j \neq i$)

Nota: Os diagramas de transição de estados são para uma dada linha de cache i.

Figura 16.20 Dois protocolos de coerência de cache.

apêndice

A

LÓGICA DIGITAL

A operação de computadores digitais é baseada no armazenamento e processamento de dados binários. Ao longo deste livro, assumimos a existência de elementos de memória, que podem estar em um de dois estados estáveis, e de circuitos capazes de operar sobre dados binários, sob controle de sinais de controle, para implementar as várias funções de um computador. Neste apêndice, sugerimos como esses elementos de memória e esses circuitos podem ser implementados em lógica digital, especificamente como circuitos combinatórios e seqüenciais. O apêndice começa com uma breve revisão de álgebra booleana, que constitui a fundamentação matemática para a lógica digital. Em seguida, introduz o conceito de porta lógica. Finalmente, são descritos circuitos combinatórios e seqüenciais, que são construídos a partir de portas lógicas.

A.1 ÁLGEBRA BOOLEANA

Os circuitos digitais de computadores e outros sistemas digitais são projetados e têm seu comportamento analisado, em termos de uma disciplina matemática conhecida como *álgebra booleana*. Esse nome é uma homenagem ao matemático inglês George Boole, que propôs os princípios básicos dessa álgebra em 1854, em seu tratado *An Investigation of the Laws of Thought on Which to Found Mathematical Theories of Logic and Probabilities*. Em 1938, Claude Shannon, um assistente de pesquisa do Departamento de Engenharia Elétrica do MIT (Massachusetts Institute of Technology), sugeriu que a álgebra booleana poderia ser usada para solucionar problemas relativos ao projeto de circuitos de comutação de relés (Shannon, 1938). As técnicas sugeridas por Shannon foram subseqüentemente usadas na análise e projeto de circuitos eletrônicos digitais. A álgebra booleana tornou-se uma ferramenta conveniente em duas áreas:

- **Análise**: ela constitui uma forma econômica de descrever a função de um circuito digital.
- **Projeto**: dada uma função a ser implementada, a álgebra booleana pode ser usada para desenvolver uma implementação simplificada para essa função.

Assim como qualquer álgebra, a álgebra booleana faz uso de variáveis e operações; nesse caso, variáveis e operações lógicas. Portanto, uma variável pode ter o valor 1 (Verdadeiro) ou 0 (Falso). As operações lógicas básicas são AND (E), OR (OU) e NOT (NÃO), que são simbolicamente representadas, respectivamente, por •, + e uma barra horizontal sobre o operando:

$$\text{A AND B} = A \bullet B$$
$$\text{A OR B} = A + B$$
$$\text{NOT A} = \overline{A}$$

A operação AND tem como resultado o valor verdadeiro (valor binário 1) se e somente se ambos os operandos têm valor verdadeiro. A operação OR tem como resultado verdadeiro, se qualquer dos operandos, ou ambos, têm valor verdadeiro. A operação unária NOT inverte o valor do operando. Por exemplo, considere a seguinte equação

$$D = A + (\overline{B} \bullet C)$$

D é igual a 1 se A é 1 ou se B = 0 e C = 1. Caso contrário, D é igual a 0.

Tabela A.1 Operadores booleanos

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P NAND Q	P NOR Q
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	0	1	1	0	0	0

Vários aspectos relativos a notação são relevantes. Na ausência de parênteses, a operação AND tem precedência sobre a operação OR. Além disso, quando não há ambigüidade, a operação AND é representada pela simples concatenação dos operandos, omitindo-se o símbolo •. Portanto,

$$A + B \bullet C = A + (B \bullet C) = A + BC$$

que significa: execute a operação AND com B e C; depois execute a operação OR com o resultado e A.

A Tabela A.1 define as operações lógicas básicas sob a forma conhecida como *tabela verdade* ou *tabela da verdade*, que simplesmente lista o valor da operação para cada possível combinação dos valores dos operandos. A tabela lista também três outros operadores úteis: XOR, NAND e NOR. A operação XOR efetua a operação de OU-Exclusivo de dois operandos, resultando em valor 1 se e somente se exatamente um dos operandos tem valor 1. A função NAND é o complemento (NOT) da função AND e a função NOR é o complemento de OR:

$$A \text{ NAND } B = \text{NOT } (A \text{ AND } B) = \overline{AB}$$

$$A \text{ NOR } B = \text{NOT } (A \text{ OR } B) = \overline{A + B}$$

Como veremos, esses três novos operadores são úteis na implementação de certos circuitos digitais.

A Tabela A.2 relaciona as principais identidades da álgebra booleana. As equações foram arranjadas em duas colunas, para evidenciar a natureza complementar, ou dual, das operações AND e OR. Existem duas classes de identidades: regras básicas (ou *postulados*), que são estabelecidas sem prova, e outras identidades que podem ser derivadas a partir dos postulados básicos. Os postulados definem como expressões booleanas são interpretadas. É importante notar que uma das duas leis de distributividade difere do que teríamos na álgebra ordinária:

$$A + (B \bullet C) = (A + B) \bullet (A + C)$$

As duas equações no final da Tabela A.2 são denominadas Leis ou Teorema de DeMorgan. Elas também podem ser escritas como:

$$A \text{ NOR } B = \overline{A} \text{ AND } \overline{B}$$

$$A \text{ NAND } B = \overline{A} \text{ OR } \overline{B}$$

Você poderá verificar as equações da Tabela A.2 substituindo as variáveis A, B e C por valores 0 ou 1.

Tabela A.2 Identidades básicas da álgebra booleana

	Postulados Básicos	
$A \bullet B = B \bullet A$	$A + B = B + A$	Leis de comutatividade
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$	$A + (B \bullet C) = (A + B) \bullet (A + C)$	Leis de distributividade
$1 \bullet A = A$	$0 + A = A$	Elemento identidade
$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$	Elemento inverso
	Outras Identidades	
$0 \bullet A = 0$	$1 + A = 1$	
$A \bullet A = A$	$A + A = A$	
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$	$A + (B + C) = (A + B) + C$	Leis de associatividade
$\overline{A \bullet B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \bullet \overline{B}$	Teorema de DeMorgan

A.2 PORTAS LÓGICAS

O bloco fundamental de construção de circuitos lógicos digitais é a porta lógica. Funções lógicas são implementadas pela conexão de portas lógicas.

Uma porta lógica é um circuito eletrônico que produz um sinal de saída que é o resultado de uma operação booleana sobre os seus sinais de entrada. As portas lógicas básicas em um circuito lógico digital são AND, OR, NOT, NAND e NOR. A Figura A.1 mostra essas cinco portas lógicas. Cada porta é definida de três formas: um símbolo gráfico, uma notação algébrica e uma tabela verdade. A simbologia usada aqui, e ao longo de todo este apêndice, corresponde ao padrão IEEE 91. Note que a operação de inversão (NOT) é indicada por um círculo.

Cada porta tem uma ou duas entradas e uma saída. Quando os valores na entrada são alterados, o sinal de saída correto aparece quase instantaneamente, sendo retardado apenas pelo tempo de propagação de sinais através da porta (conhecido como *atraso da porta lógica*). A importância disso é discutida na Seção A.3.

Além das portas mostradas na Figura A.1, também podem ser usadas portas com três, quatro ou mais entradas. Assim, por exemplo, X + Y + Z pode ser implementado como uma porta lógica OR com três entradas.

Tipicamente, nem todos os tipos de porta são usados em uma implementação. O projeto e a fabricação de circuitos lógicos tornam-se mais simples se são usados apenas um ou dois tipos de porta. Portanto, é importante identificar que conjuntos de portas lógicas são *funcionalmente completos*. Isso significa que qualquer função booleana pode ser implementada usando apenas as portas desse conjunto. Os seguintes conjuntos de portas são funcionalmente completos:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR

Nome	Símbolo gráfico	Função algébrica	Tabela verdade
AND	A B F	F = A • B ou F = AB	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A B F	F = A + B	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT	A F	$F = \overline{A}$ ou F = A'	A | F 0 | 1 1 | 0
NAND	A B F	$F = (\overline{AB})$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A B F	$F = (\overline{A + B})$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

Figura A.1 Portas lógicas básicas.

Deve ficar claro que as portas AND, OR e NOT constituem também um conjunto funcionalmente completo, uma vez que representam as três operações da álgebra booleana. Para que as portas AND e NOT constituam um conjunto funcionalmente completo, deve existir uma maneira de expressar a operação OR usando as operações AND e NOT. Isso pode ser obtido usando as Leis de DeMorgan:

$$A + B = \overline{\overline{A} \bullet \overline{B}}$$

A OR B = NOT ((NOT A) AND (NOT B))

Analogamente, o conjunto formado pelas operações OR e NOT é funcionalmente completo, pois essas operações podem ser usadas para sintetizar a operação AND.

A Figura A.2 mostra como as funções AND, OR e NOT podem ser implementadas usando apenas portas NAND, e a Figura A.3 mostra como essas funções podem ser implementadas usando apenas portas NOR. Por essa razão, circuitos digitais podem ser, e freqüentemente são, implementados apenas com portas NAND ou apenas com portas NOR.

Com as portas lógicas, atingimos o nível mais primitivo da ciência e engenharia de computação. Um exame das combinações de transistores usadas para construir portas lógicas foge ao escopo da ciência da computação e da engenharia de computação, entrando na área da engenharia elétrica. Para o nosso propósito, é suficiente descrever como as portas lógicas são usadas como blocos básicos na implementação dos circuitos lógicos essenciais de um computador digital.

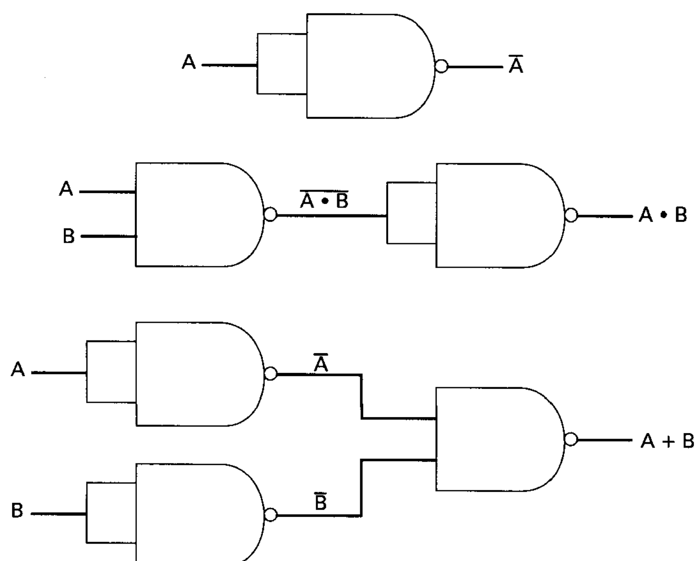


Figura A.2 O uso de portas NAND.

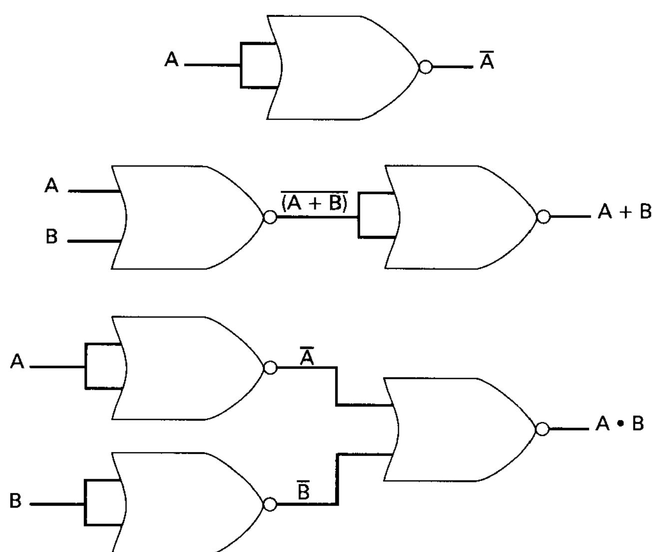


Figura A.3 O uso de portas NOR.

A.3 CIRCUITOS COMBINATÓRIOS

Um circuito combinatório consiste em uma interconexão de portas lógicas, cujo sinal de saída é, em qualquer instante, função apenas dos seus sinais de entrada naquele instante. Assim como em uma porta simples, a alteração de sinais de entrada é quase imediatamente seguida pela alteração correspondente no sinal de saída, apenas com retardo devido à transmissão de sinais por meio das portas do circuito.

Em termos gerais, um circuito combinatório consiste de n entradas binárias e m saídas binárias. Assim como uma porta, um circuito combinatório pode ser definido de três formas:

- **Tabela verdade**: para cada uma das possíveis combinações dos n sinais de entrada, é listado o valor binário de cada um dos m sinais de saída.
- **Símbolos gráficos**: esquema de conexão das portas lógicas.
- **Equações booleanas**: cada sinal de saída é expresso como uma função booleana dos sinais de entrada.

Implementação de funções booleanas

Qualquer função booleana pode ser implementada por um circuito eletrônico, na forma de uma rede de portas lógicas. Para qualquer função, existem diversas implementações alternativas. Considere a função booleana representada pela tabela verdade da Tabela A.3. Podemos expressar essa função simplesmente relacionando as combinações de valores das variáveis A, B e C que tornam F igual a 1:

$$F = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} \qquad (A.1)$$

Existem três possíveis combinações dos valores de entrada que fazem F igual a 1; se qualquer dessas combinações ocorrer, o resultado na saída será 1. Essa forma de expressar é denominada, por razões evidentes, uma *soma de produtos* (SOP). A Figura A.4 mostra uma implementação direta da função F, usando portas AND, OR e NOT.

Tabela A.3 Função booleana com três variáveis

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

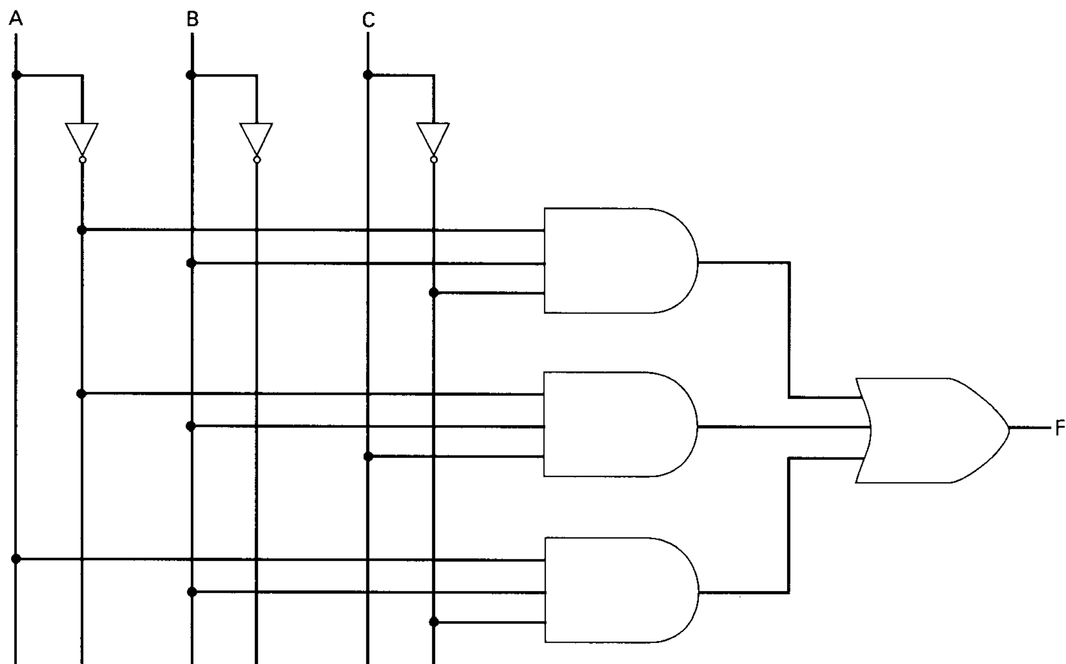


Figura A.4 Implementação em soma de produtos da Tabela A.3.

Outra forma pode também ser derivada da tabela verdade. A forma SOP expressa que a saída é igual a 1 se for verdade que alguma das combinações de entradas produz 1. Podemos também dizer que a saída é 1 se for falso que nenhuma das combinações de entrada produz 0. Portanto,

$$F = \overline{(\overline{A}\,\overline{B}\,\overline{C})} \bullet \overline{(\overline{A}\,\overline{B}C)} \bullet \overline{(A\overline{B}\,\overline{C})} \bullet \overline{(A\overline{B}C)} \bullet \overline{(ABC)}$$

Isso pode também ser escrito, usando uma generalização da Lei de DeMorgan:

$$\overline{X \bullet Y \bullet Z} = \overline{X} + \overline{Y} + \overline{Z}$$

Portanto,

$$\begin{aligned} F &= (\overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}}) \bullet (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \bullet (\overline{A} + \overline{\overline{B}} + \overline{\overline{C}}) \bullet (\overline{A} + \overline{\overline{B}} + \overline{C}) \bullet (\overline{A} + \overline{B} + \overline{C}) \\ &= (A + B + C) \bullet (A + B + \overline{C}) \bullet (\overline{A} + B + C) \bullet (\overline{A} + B + \overline{C}) \bullet (\overline{A} + \overline{B} + \overline{C}) \end{aligned} \quad \text{(A.2)}$$

Nesse caso, F é expresso na forma de um *produto de somas* (POS), que é ilustrado na Figura A.5. Para maior clareza, não mostramos as portas NOT, e consideramos que cada sinal de entrada e seu complemento estão disponíveis. Isso simplifica o diagrama lógico e torna mais aparentes as entradas para cada porta lógica.

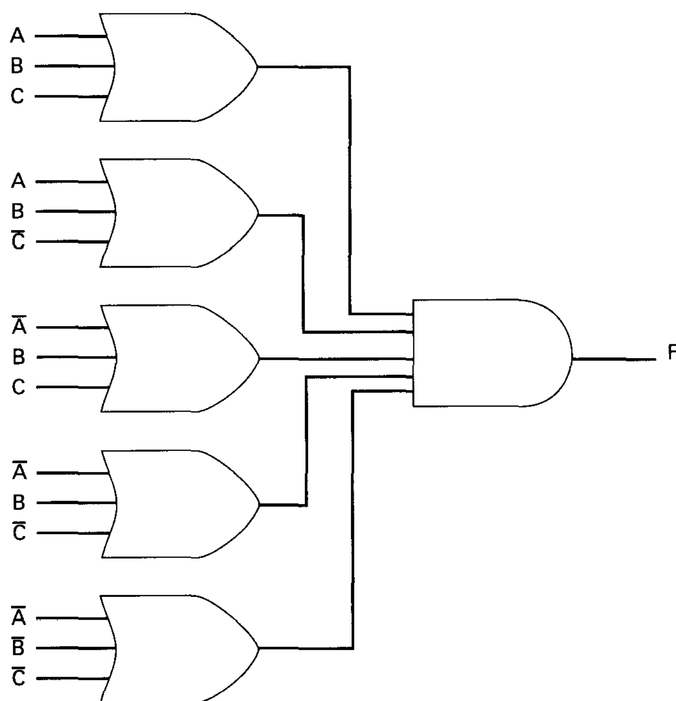


Figura A.5 Implementação em produto de somas da Tabela A.3.

Uma função booleana pode, portanto, ser expressa tanto como uma soma de produtos quanto como um produto de somas. Até aqui, parece que a escolha entre uma ou outra forma depende de se a saída da função, na tabela verdade, contém maior número de 1s ou de 0s: a forma SOP tem um termo para cada 1 e a forma POS tem um termo para cada 0. Entretanto, existem outros pontos a considerar:

- Geralmente é possível derivar uma expressão booleana mais simples a partir da tabela verdade do que a expressão equivalente na forma SOP ou POS.
- Pode ser preferível implementar a função com um único tipo de porta (NAND ou NOR).

O significado do primeiro ponto acima é que, se uma expressão booleana é dada em uma forma mais simples, um menor número de portas é requerido para implementar a função. Três métodos podem ser usados para simplificar expressões booleanas:

- Simplificação algébrica
- Mapas de Karnaugh
- Tabelas de Quine-McKluskey

Simplificação algébrica

A simplificação algébrica envolve a aplicação das identidades relacionadas na Tabela A.2, para reduzir uma expressão booleana a uma outra com menor número de elementos. Por

exemplo, considere novamente a Equação (A.1). Com um pouco de trabalho, você poderá verificar que ela é equivalente à expressão:

$$F = \overline{A}B + B\overline{C} \quad \text{(A.3)}$$

ou mesmo à expressão, ainda mais simples:

$$F = B(\overline{A} + \overline{C})$$

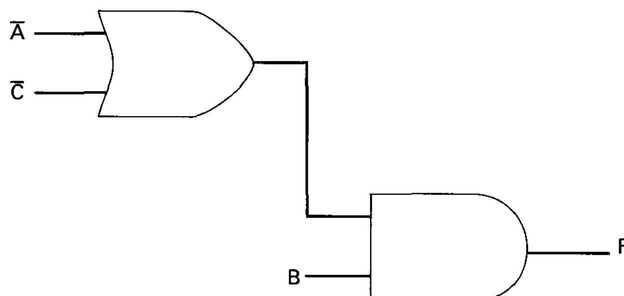


Figura A.6 Implementação simplificada da Tabela A.3.

Essa expressão pode ser implementada como mostrado na Figura A.6. A simplificação da Equação (A.1) pode ser feita essencialmente por observação. Para expressões mais complexas, é necessário adotar uma abordagem sistemática.

Mapas de Karnaugh

O mapa de Karnaugh é uma forma de representação conveniente de uma função booleana com pequeno número de variáveis (até 4 ou 6), visando à simplificação da expressão que define essa função. O mapa consiste em uma matriz de posições, representando as possíveis combinações de valores de *n* variáveis binárias. A Figura A.7a mostra um mapa de quatro posições para uma função de duas variáveis. É conveniente, como veremos mais adiante, listar essas combinações na ordem 00, 01, 11, 10. Como as posições correspondentes às combinações são usadas para registrar informações, as combinações usualmente são escritas acima de cada posição. No caso de três variáveis, a representação é um arranjo de oito posições (Figura A.7b), com os valores de uma das variáveis à esquerda e os valores das duas outras variáveis acima de cada posição. Para quatro variáveis, são requeridas 16 posições, arranjadas tal como indicado na Figura A.7c.

O mapa de Karnaugh pode ser usado para representar qualquer função booleana, da seguinte forma. Cada posição corresponde a um único produto da expressão na forma de soma de produtos, onde o valor 1 corresponde à variável e o valor 0 corresponde à negação (NOT) dessa variável. Portanto, o produto $A\overline{B}$ corresponde à quarta posição na Figura A.7a. Para cada um dos produtos da expressão que define a função, é colocado o valor 1 na posição correspondente. Portanto, no exemplo com duas variáveis, o mapa corresponde à função definida pela expressão $A\overline{B} + \overline{A}B$. Dada a tabela verdade de uma função booleana, é fácil construir o mapa de Karnaugh correspondente: para cada combinação de valores das variáveis que produz resultado 1 na tabela verdade, preencha a posição correspondente com valor 1. A Figura A.7b mostra o mapa correspondente à tabela verdade da Tabela A.3. Para converter

uma expressão booleana para um mapa, é necessário primeiro escrever essa expressão no que se chama *forma canônica*: cada termo da expressão deve conter cada variável. Assim, por exemplo, para a Equação (A.3), primeiro teríamos de expandi-la para a forma completa dada pela Equação (A.1), e então obter o mapa correspondente.

Os rótulos mostrados na Figura A.7d enfatizam o relacionamento entre as variáveis e as linhas e colunas do mapa. Nesse caso, as duas linhas indicadas pelo rótulo A são aquelas em que a variável A tem valor 1; as linhas que não têm o rótulo A são aquelas em que a variável A tem valor 0 (analogamente para B, C e D).

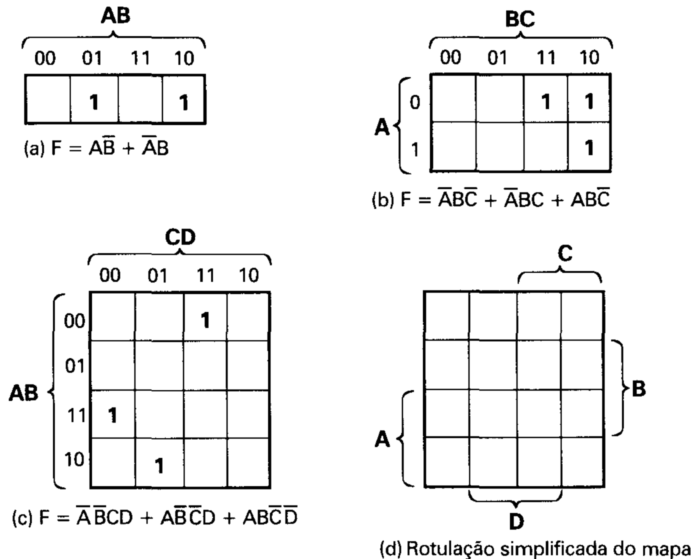


Figura A.7 Uso de mapas de Karnaugh para representar funções booleanas.

Uma vez criado o mapa da função, podemos quase sempre escrever uma expressão algébrica equivalente mais simples, verificando o arranjo de 1s do mapa. O princípio é o seguinte: quaisquer duas posições adjacentes diferem em apenas uma das variáveis; se duas posições adjacentes têm ambas valor igual a 1, então os termos correspondentes do produto diferem em apenas uma variável. Nesse caso, os dois termos podem ser combinados, eliminando essa variável. Por exemplo, na Figura A.8a, as duas posições adjacentes correspondem aos termos $\overline{A}B\overline{C}D$ e $\overline{A}BCD$. Portanto, a função pode ser expressa como

$$\overline{A}B\overline{C}D + \overline{A}BCD = \overline{A}BD$$

Esse processo pode ser estendido de várias formas. Primeiro, o conceito de adjacência pode ser estendido para incluir um giro em torno das extremidades do mapa. Assim, a posição no topo de uma coluna é adjacente à posição mais embaixo da mesma coluna; a posição mais à esquerda de uma linha é adjacente à posição mais à direita da mesma linha. Essas situações são mostradas na Figura A.8b e A.8c. Segundo, podemos agrupar não apenas duas, mas posições adjacentes; isto é, 4, 8 etc. Os três exemplos seguintes da Figura A.8 mostram agrupamentos de 4 posições. Note que, nesse caso, duas das variáveis podem ser eliminadas.

Os três últimos exemplos mostram agrupamentos de oito posições, que possibilitam eliminar três variáveis.

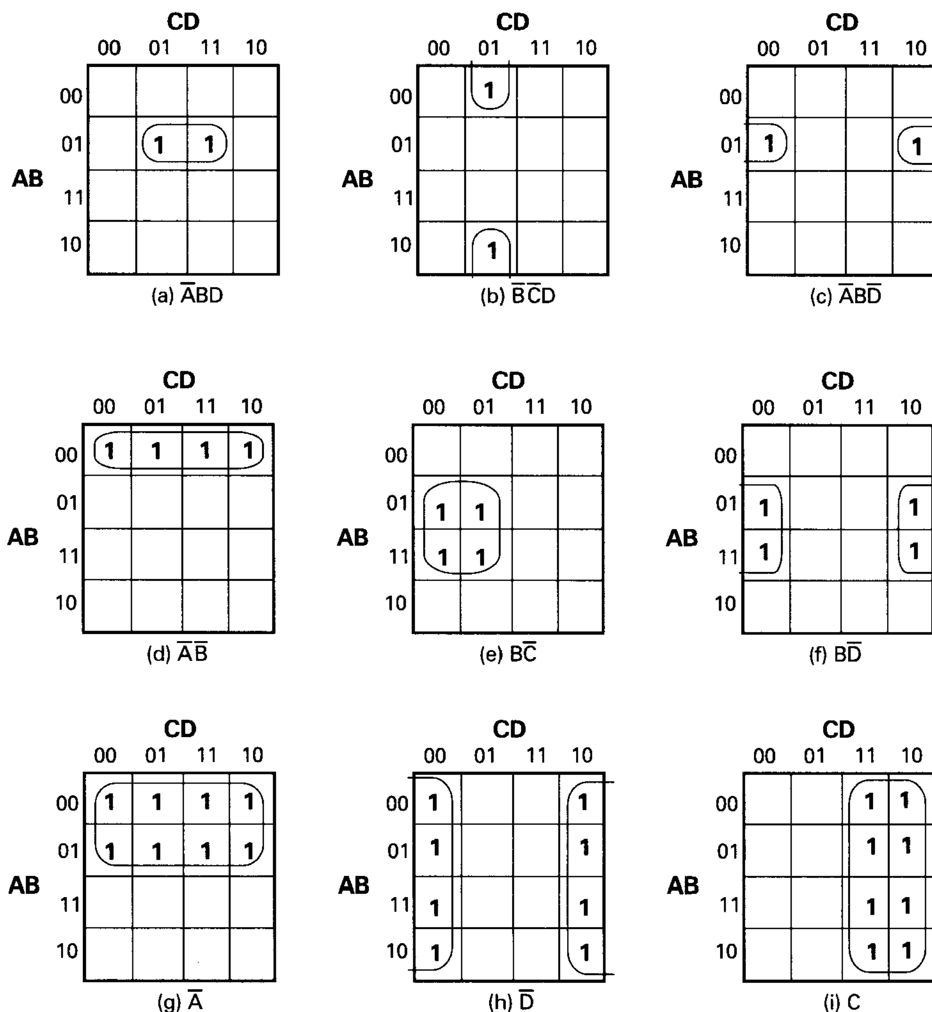


Figura A.8 Uso de mapas de Karnaugh.

Podemos resumir as regras de simplificação do seguinte modo:

1. Dentre as posições marcadas (posições com valor 1), encontre aquelas que pertencem a um único bloco máximo de 1, 2, 4 ou 8 posições e contorne esses blocos.
2. Selecione blocos adicionais de posições marcadas, que sejam tão grandes quanto possível e em menor número possível, mas que incluam toda posição marcada, pelo menos uma vez. Pode não haver um resultado único em alguns casos. Por exemplo, se uma posição marcada combina exatamente com duas outras posições, e não existe uma quarta posição marcada para completar um bloco maior, então existem duas possibilidades de

escolha de agrupamento. Quando se estiver marcando blocos, é permitido usar a mesma posição de valor 1 mais de uma vez.

3. Continue marcando o contorno de posições marcadas simples, ou pares de posições marcadas adjacentes, ou grupos de quatro, oito etc. posições marcadas adjacentes, de modo que cada posição marcada pertença a pelo menos um bloco; então use o menor número possível desses blocos que inclua todas as posições marcadas.

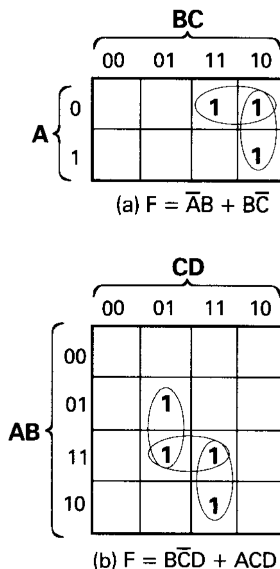


Figura A.9 Grupos sobrepostos.

A Figura A.9a, baseada na Tabela A.3, ilustra esse procedimento. Se restar alguma posição marcada isolada, depois do agrupamento em blocos, então cada uma delas será marcada como um bloco. Finalmente, antes de converter o mapa em uma expressão booleana simplificada, qualquer bloco de 1s que seja completamente sobreposto por outros blocos é eliminado. Isso é mostrado na Figura A.9b. Nesse caso, o grupo horizontal é redundante, e pode ser ignorado na construção da expressão booleana.

Uma característica adicional dos mapas de Karnaugh deve ser mencionada. Em alguns casos, certas combinações de valores de variáveis nunca ocorrem e, portanto, a saída correspondente nunca ocorre. Esses casos são denominados "negligenciáveis" (*don't care*). Para cada um deles, é usada a letra "d" na posição correspondente no mapa. Ao fazer o agrupamento de posições marcadas e a simplificação, cada "d" pode ser tratado como um valor 1 ou 0, escolhendo-se o valor que resulta na expressão mais simples.

Um exemplo, apresentado em Hayes (1988), ilustra os pontos que foram discutidos. Suponha que queremos obter uma expressão booleana para um circuito lógico que soma 1 a um dígito decimal empacotado. Lembre-se, da Seção 9.2, que na representação de número decimal empacotado cada dígito decimal é representado por um código com 4 bits, da forma óbvia. Assim, 0 = 0000, 1 = 0001, ... , 8 = 1000 e 9 = 1001. Os demais valores representáveis com 4 bits, de 1010 a 1111, não são usados. Esse código é também chamado de BCD (binary coded decimal).

A Tabela A.4 mostra a tabela verdade para produzir um resultado de 4 bits que é dado por um dígito BCD fornecido como entrada mais 1. A adição é módulo 10, isto é, 9 + 1 = 0. Além disso, note que seis dos possíveis códigos de entrada produzem resultado "negligenciável", pois não correspondem a dígitos BCD válidos. A Figura A.10 mostra o mapa de Karnaugh resultante para cada uma das variáveis de saída. O valor atribuído às posições marcadas com "d" é escolhido de modo que obtenha os agrupamentos que resultam em maior simplificação da expressão.

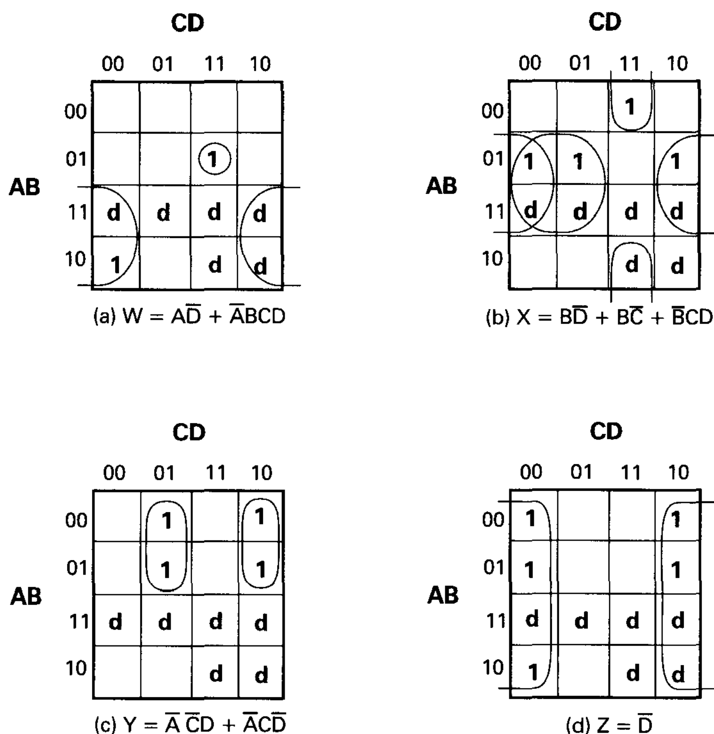


Figura A.10 Mapas de Karnaugh para o incrementador BCD.

Tabela A.4 Tabela verdade para função que incrementa um dígito BCD

Entrada					Saída				
Número	**A**	**B**	**C**	**D**	**Número**	**W**	**X**	**Y**	**Z**
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	2	0	0	1	0
2	0	0	1	0	3	0	0	1	1
3	0	0	1	1	4	0	1	0	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	6	0	1	1	0
6	0	1	1	0	7	0	1	1	1

Tabela A.4 Tabela verdade para função que incrementa um dígito BCD (*continuação*)

	Entrada				Saída				
Número	**A**	**B**	**C**	**D**	**Número**	**W**	**X**	**Y**	**Z**
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	9	1	0	0	1
9	1	0	0	1	0	0	0	0	0
Casos "negligenciáveis"	1	0	1	0		d	d	d	d
	1	0	1	1		d	d	d	d
	1	1	0	0		d	d	d	d
	1	1	0	1		d	d	d	d
	1	1	1	0		d	d	d	d
	1	1	1	1		d	d	d	d

O método de Quine-McKluskey

Para simplificar expressões com mais de quatro variáveis, o método do mapa de Karnaugh torna-se cada vez mais complicado. Com cinco variáveis, são necessários dois mapas de 16 × 16, sendo um mapa considerado como se estivesse ao topo de outro, em três dimensões, para fins de determinar adjacência entre as posições. Seis variáveis requerem o uso de quatro mapas de 16 × 16, em quatro dimensões! Uma abordagem alternativa é uma técnica tabular, conhecida como método de Quine-McKluskey. O método é adequado para ser programado em um computador, provendo uma ferramenta automática para minimizar expressões booleanas.

O método é mais facilmente explicado por meio de um exemplo. Considere a seguinte expressão:

$$ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$$

Suponha que essa expressão foi derivada de uma tabela verdade. Gostaríamos de obter uma expressão mínima equivalente, adequada para ser implementada usando portas lógicas.

O primeiro passo é construir uma tabela na qual cada linha corresponde a um dos termos produto da expressão. Os termos são agrupados de acordo com o número de variáveis negadas. Isto é, começamos com o termo em que não ocorre nenhuma variável negada, caso exista; então todos os termos com uma variável negada, e assim por diante. A Tabela A.5 mostra essa lista, no caso da expressão tomada como exemplo, onde cada linha horizontal é usada para indicar um agrupamento. Para maior clareza, cada termo é representado usando valor 1, para cada variável não negada, e valor 0, para cada variável negada. Assim, os termos são agrupados de acordo com o número de 1s que eles contêm. A coluna de índices contém o valor decimal equivalente e é útil na discussão a seguir.

Tabela A.5 Primeiro estágio do método de Quine-McKluskey
(para F = $ABCD + AB\bar{C}\bar{D} + AB\bar{C}\bar{D} + A\bar{B}CD + \bar{A}BCD + \bar{A}BC\bar{D} + \bar{A}B\bar{C}D + \bar{A}\bar{B}\bar{C}D$)

Termo produto	Índice	A	B	C	D	
$\bar{A}\bar{B}\bar{C}D$	1	0	0	0	1	✔
$\bar{A}B\bar{C}D$	5	0	1	1	1	✔
$\bar{A}BC\bar{D}$	6	0	1	1	0	✔
$AB\bar{C}\bar{D}$	12	1	1	0	0	✔
$\bar{A}BCD$	7	0	1	1	1	✔
$A\bar{B}CD$	11	1	0	1	1	✔
$AB\bar{C}D$	13	1	1	0	1	✔
ABCD	15	1	1	1	1	✔

O passo seguinte é encontrar todos os pares de termos que diferem em apenas uma variável (ou seja, todos os pares de termos que são iguais, exceto que uma das variáveis tem valor 0 em um deles e valor 1 no outro). Devido à maneira como os termos foram agrupados, isso pode ser feito começando pelo primeiro grupo e comparando cada termo do primeiro grupo com todos os termos do segundo grupo. Em seguida, comparamos cada termo do segundo grupo com todos os termos do terceiro grupo, e assim por diante. Sempre que ocorrer um emparelhamento (*match*), colocamos uma marca próxima a cada termo, combinamos o par eliminando a variável que difere nos dois termos e adicionamos o resultado a uma nova lista. Assim, por exemplo, os termos ABCD e $AB\bar{C}D$ são combinados para formar ABD. Esse processo continua até que toda a tabela original tenha sido examinada. O resultado é uma nova tabela, com as seguintes entradas:

$\bar{A}\bar{C}D$	$AB\bar{C}$	ABD ✔
	$B\bar{C}D$ ✔	ACD
	$\bar{A}BC$	BCD ✔
	$\bar{A}BD$ ✔	

A nova tabela é organizada em grupos, tal como indicado, da mesma forma que a primeira tabela. A segunda tabela é então processada da mesma maneira que a primeira. Isto é, verifica-se quais os termos que diferem em apenas uma variável, e então estes são combinados, produzindo um termo para uma terceira tabela. Nesse exemplo, a terceira tabela produzida contém apenas um elemento: BD.

Em geral, o processo prossegue por várias tabelas sucessivas, até que seja produzida uma tabela em que não ocorre nenhum emparelhamento entre termos. Nesse caso, isso envolveu três tabelas.

Uma vez que esse processo é completado, teremos eliminado a maior parte dos termos da expressão passíveis de serem eliminados. Os termos que não foram eliminados são usados para construir uma matriz, tal como ilustrado na Tabela A.6. Cada linha da matriz corresponde a um dos termos que não foram eliminados (não marcado) em qualquer das tabelas produzidas até então. Cada coluna corresponde a um dos termos da expressão original.

Colocamos um X em cada interseção de uma linha e uma coluna tal que o elemento da linha seja "compatível" com o elemento da coluna. Isto é, as variáveis presentes no elemento da linha devem ter o mesmo valor que as variáveis presentes no elemento da coluna. Em seguida marcamos com um círculo cada X que ocorra sozinho em uma coluna. Então, marcamos com um quadrado cada X que ocorra em uma linha onde existe um X marcado com um círculo. Se toda coluna tiver um X marcado com um círculo ou com um quadrado, então teremos terminado, e os elementos das linhas onde exista um X marcado constituem a expressão mínima. Portanto, no nosso exemplo, a expressão final é:

$$AB\overline{C} + ACD + \overline{A}BC + \overline{A}\,\overline{C}D$$

Tabela A.6 Último estágio do método de Quine-McKluskey
(para $F = ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$

	$ABCD$	$AB\overline{C}D$	$AB\overline{C}\,\overline{D}$	$A\overline{B}CD$	$\overline{A}BCD$	$\overline{A}BC\overline{D}$	$\overline{A}B\overline{C}D$	$\overline{A}\,\overline{B}\,\overline{C}D$
BD	X	X			X		X	
$\overline{A}\,\overline{C}D$							[X]	⊗
$\overline{A}BC$					[X]	⊗		
$AB\overline{C}$		[X]	⊗					
ACD	[X]			⊗				

Nos casos em que algumas colunas não têm nenhum X marcado com um círculo ou com um quadrado, é requerido processamento adicional. Essencialmente, continuamos a adicionar elementos nas linhas, até que todas as colunas sejam cobertas.

Vamos procurar resumir o método de Quine-McKluskey e tentar justificar, intuitivamente, porque esse método funciona. A primeira fase do método é razoavelmente direta. O processo elimina variáveis desnecessárias nos termos produto. Assim, a expressão $ABC + AB\overline{C}$ é equivalente a AB, pois

$$ABC + AB\overline{C} = AB(C + \overline{C}) = AB$$

Depois de eliminar essas variáveis, obtemos uma expressão que, claramente, é equivalente à expressão original. Entretanto, podem existir termos redundantes na expressão, do mesmo modo como encontramos agrupamentos redundantes usando mapas de Karnaugh. O esquema da matriz assegura que cada termo da expressão original seja coberto, e faz isso para minimizar o número de termos da expressão final.

Implementações usando portas NAND e NOR

Outra consideração na implementação de funções booleanas se refere aos tipos de portas usadas. Freqüentemente, é desejável implementar uma função booleana usando apenas a porta NAND ou apenas a porta NOR. Embora isso possa não ser a implementação que utiliza o menor número de portas, ela tem como vantagem a regularidade, o que pode simplificar o processo de fabricação. Considere novamente a Equação (A.3):

$$F = B(\overline{A} + \overline{C})$$

Como o complemento do complemento de um valor é exatamente o valor original, temos:

$$F = B(\overline{A} + \overline{C}) = (\overline{A}B) + (B\overline{C})$$

Aplicando a Lei de DeMorgan, temos:

$$F = \overline{\overline{(\overline{A}B)} \bullet \overline{(B\overline{C})}}$$

que tem três portas NAND, como ilustrado na Figura A.11.

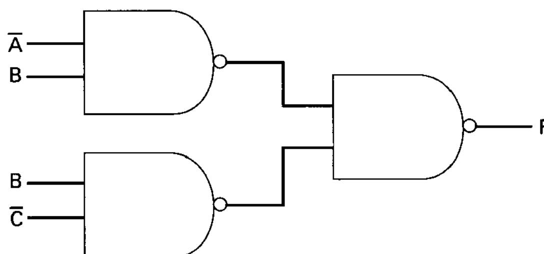


Figura A.11 Implementação da Tabela A.3 usando portas NAND.

Multiplexadores

Um multiplexador (MUX) conecta várias entradas em uma única saída. Em qualquer instante, é selecionada uma das entradas, para ser passada para a saída. Uma representação na forma de diagrama de blocos genérico é mostrada na Figura A.12. Ela representa um multiplexador 4-para-1. Existem quatro linhas de entrada, identificadas por D0, D1, D2 e D3. Uma dessas linhas é selecionada para fornecer o sinal de saída F. Um código de seleção de 2 bits é usado para selecionar uma das quatro linhas de entrada, e é implementado pelas duas linhas de seleção, identificadas por S1 e S2.

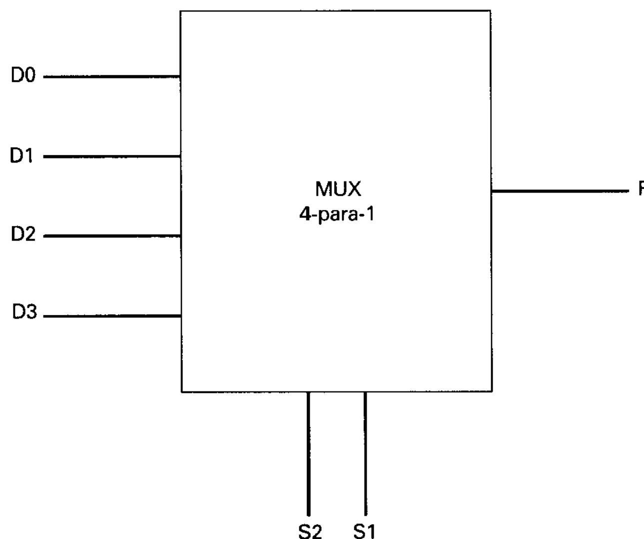


Figura A.12 Representação de um multiplexador (MUX) 4-para-1.

Um exemplo de multiplexador 4-para-1 é definido pela tabela verdade da Tabela A.7. Essa é uma forma simplificada de tabela verdade: em vez de mostrar todas as possíveis combinações das variáveis de entrada, ela mostra a saída como sendo o dado contido nas linhas D0, D1, D2 ou D3. A Figura A.13 mostra uma implementação desse multiplexador usando portas AND, OR e NOT. As linhas S1 e S2 são conectadas a portas AND de tal modo que, para cada combinação de S1 e S2, três das portas AND terão como saída o valor 0. A quarta porta AND terá como saída o valor da linha selecionada, que pode ser 0 ou 1. Portanto, três das entradas para a porta OR serão sempre 0, e a saída da porta OR será igual ao valor da linha de entrada selecionada. Usando essa organização regular, é fácil construir multiplexadores com tamanho 8-para-1, 16-para-1, e assim por diante.

Tabela A.7 Tabela verdade do multiplexador 4-para-1

S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

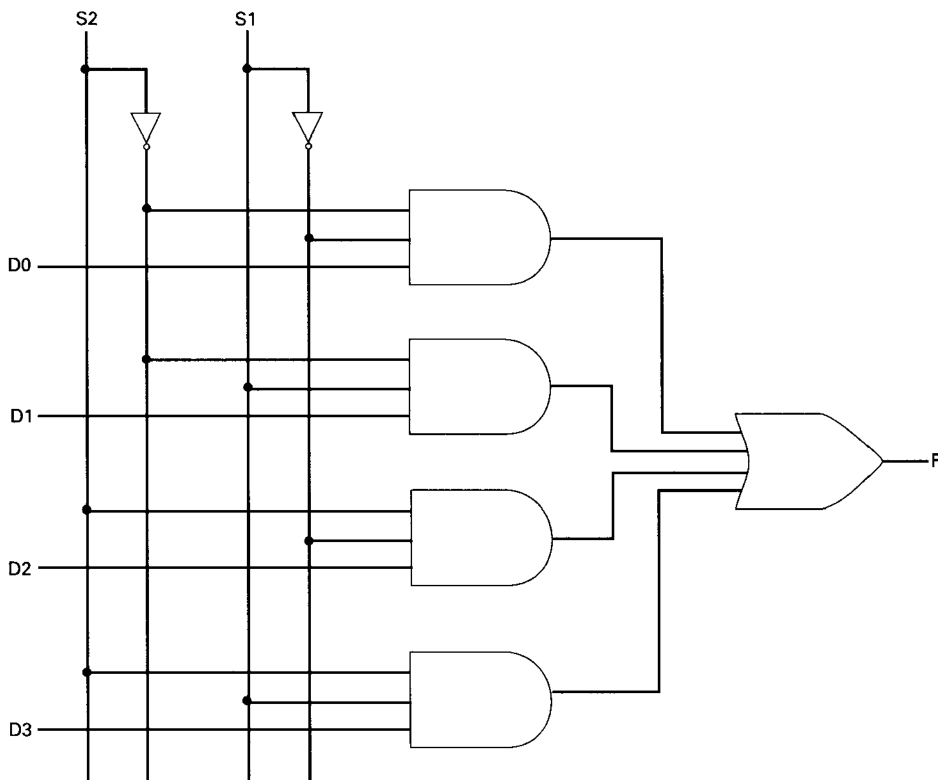


Figura A.13 Implementação do multiplexador 4-para-1.

Multiplexadores são usados em circuitos digitais para controle e roteamento de sinais. Um exemplo é a carga de um valor no contador de programa (PC). O valor a ser carregado no contador de programa pode vir de diferentes fontes:

- De um contador binário, se o PC for incrementado para a próxima instrução.
- Do registrador de instrução, se foi executada uma instrução de desvio com endereçamento imediato.
- Da saída da ULA, se a instrução de desvio especifica o endereço usando modo de endereçamento por deslocamento.

Essas várias entradas poderiam ser conectadas nas entradas de um multiplexador, sendo o contador de programa conectado na saída. As linhas de seleção determinariam o valor a ser carregado no contador de programa. Como o contador de programa consiste de múltiplos bits, seriam usados vários multiplexadores, um por bit. A Figura A.14 ilustra esse esquema, para um endereço de 16 bits.

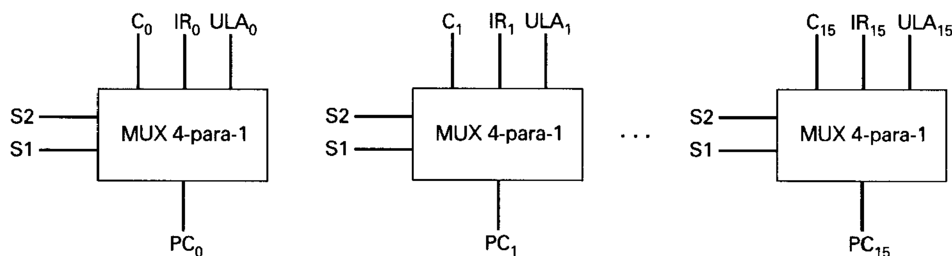


Figura A.14 Entrada de multiplexador para o contador de instruções.

Decodificadores

Um decodificador é um circuito combinatório com um certo número de linhas de saída, apenas uma das quais é ativada em cada instante, dependendo do padrão de sinais nas linhas de entrada. Em geral, um decodificador tem n entradas e 2^n saídas. A Figura A.15 mostra um decodificador com três entradas e oito saídas.

Decodificadores têm diferentes usos em um computador digital. Uma exemplo é na decodificação de endereços. Suponha que queremos construir uma memória de 1 Kbyte, usando quatro pastilhas de memória RAM de 256 × 8 bits. Desenhamos um espaço de endereçamento único, podendo ser composto do seguinte modo:

Endereço	Pastilha
0000-00FF	0
0100-01FF	1
0200-02FF	2
0300-03FF	3

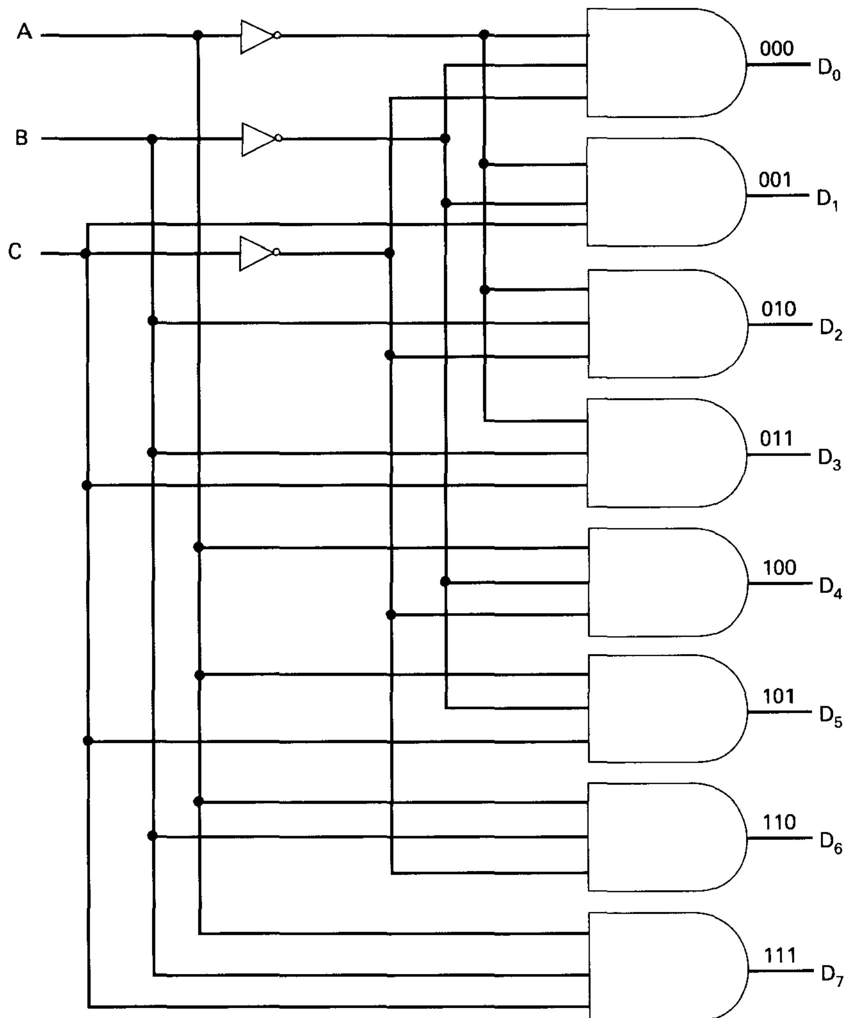


Figura A.15 Decodificador com 3 entradas e $2^3 = 8$ saídas.

Cada pastilha requer 8 linhas de endereço, que são fornecidas pelos 8 bits menos significativos do endereço. Os dois bits mais significativos do endereço de 10 bits são usados para selecionar uma das quatro pastilhas de memória RAM. Para isso, é usado um decodificador 2-para-4, cuja saída habilita uma das quatro pastilhas, como mostrado na Figura A.16.

Com uma linha de entrada adicional, o decodificador pode ser usado como um demultiplexador. O demultiplexador efetua a função inversa do multiplexador; ele conecta uma única entrada a múltiplas saídas. Isso é mostrado na Figura A.17. Assim como antes, *n* entradas são decodificadas para produzir apenas uma saída dentre as saídas. É efetuada uma operação lógica AND envolvendo todas as saídas e cada linha de entrada de dados. Assim, as *n* linhas de entrada agem como um endereço para selecionar uma linha de saída particular, e o valor da linha de entrada (0 ou 1) é roteado para essa linha de saída.

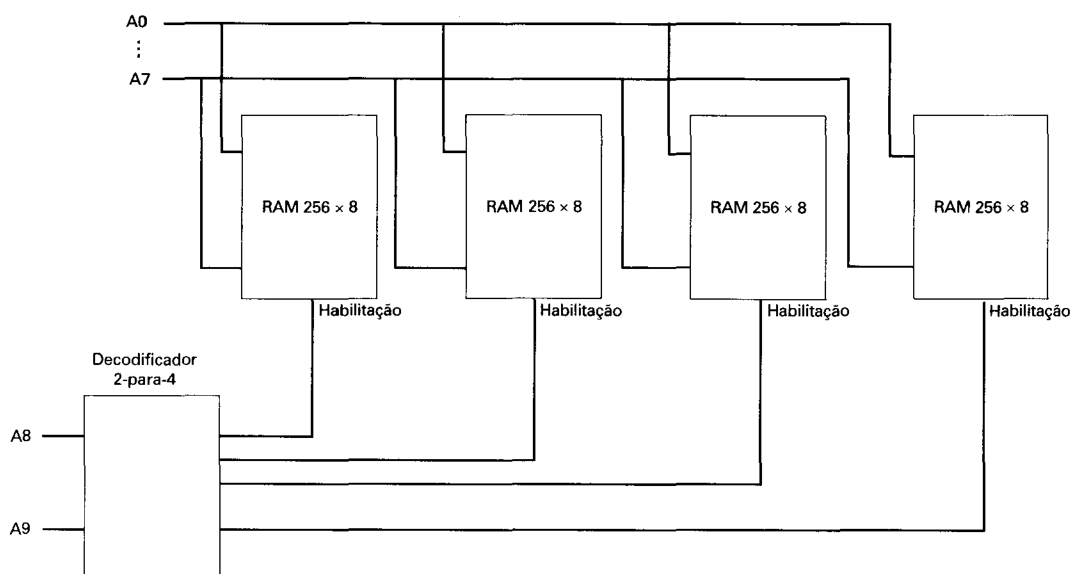


Figura A.16 Decodificação de endereços.

A configuração mostrada na Figura A.17 pode também ser vista de outra forma. Mude a identificação da linha nova, de *Entrada de Dados* para *Habilitação*. Isso possibilita o controle de temporização do decodificador. O sinal aparece na saída do decodificador apenas quando existe entrada presente *e* a linha de habilitação tem valor 1.

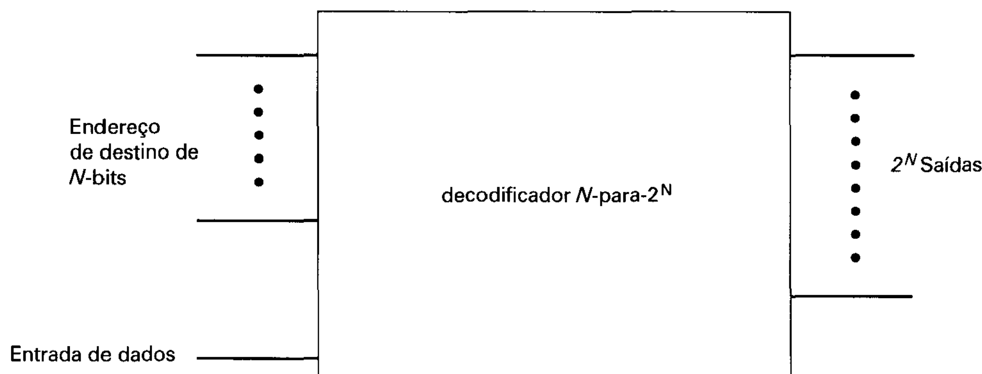


Figura A.17 Implementação de um demultiplexador usando um decodificador.

Matriz de lógica programável

Até agora, tratamos portas individuais como blocos básicos de construção de circuitos digitais, a partir dos quais qualquer função lógica pode ser implementada. O projetista poderia buscar uma estratégia para minimizar o número de portas a serem usadas, manipulando a expressão booleana correspondente.

Com o crescimento do nível de integração dos circuitos integrados, outras considerações podem ser feitas. Os primeiros circuitos integrados utilizavam integração em baixa escala (SSI — small scale integration), provendo uma a dez portas por pastilha. Na abordagem de blocos de construção descrita anteriormente, cada porta é tratada de forma independente. A Figura A.18 mostra exemplos de algumas pastilhas SSI. Para construir uma função lógica, um certo número dessas pastilhas é disposto sobre uma placa de circuito impresso e são feitas as conexões de pinos apropriadas.

Os crescentes níveis de integração possibilitaram incluir maior número de portas em uma pastilha, assim como incluir conexões entre portas na própria pastilha. Isso tem a vantagem de diminuir o custo e o tamanho, e aumentar a velocidade (porque o atraso na propagação de sinais interna a uma pastilha é menor que no caso de sinais externos à pastilha). Entretanto, surge um problema de projeto. Para cada função lógica particular, ou conjunto de funções, deve ser projetado um conjunto de portas e suas conexões na pastilha. O projeto de pastilhas específicas requer muito tempo e tem um custo altíssimo. Portanto, torna-se atrativo desenvolver pastilhas de uso geral, que possam ser facilmente adaptadas para propósitos específicos. Essa é a intenção de uma *matriz de lógica programável* (PLA — programmable logic array).

A PLA é baseada no fato de que qualquer função booleana (tabela verdade) pode ser expressa na forma de uma soma de produtos (SOP), como vimos anteriormente. Uma PLA consiste em um arranjo regular de portas NOT, AND e OR em uma pastilha. Cada entrada da pastilha passa por uma porta NOT, de forma que cada entrada e seu complemento ficam disponíveis para cada porta AND. A saída de cada porta AND é conectada à entrada de cada porta OR, e a saída de cada porta OR é uma saída da pastilha. Fazendo as conexões apropriadas, é possível implementar uma função booleana arbitrária, na forma SOP.

A Figura A.19a mostra uma PLA com três entradas, oito portas e duas saídas. As PLAs maiores têm várias centenas de portas lógicas, 15 a 25 entradas e 5 a 15 saídas. As conexões das entradas às portas AND, e das portas AND às portas OR, não são especificadas.

As PLAs podem ser fabricadas de duas formas diferentes, para tornar mais fácil a sua programação (estabelecimento das conexões). Na primeira, cada possível conexão é feita por meio de um fusível em cada ponto de interseção. As conexões não desejadas podem ser mais tarde removidas, derretendo esses fusíveis. Esse tipo de PLA é conhecido como *matriz de lógica programável de campo* (*field-programmable logic array*). Alternativamente, as conexões apropriadas podem ser feitas durante a fabricação da pastilha, usando uma máscara apropriada fornecida pelo usuário para cada padrão particular de conexões. Em qualquer dos casos, a PLA provê uma forma flexível e barata para implementar funções lógicas digitais.

A Figura A.19b mostra um projeto que implementa duas expressões booleanas.

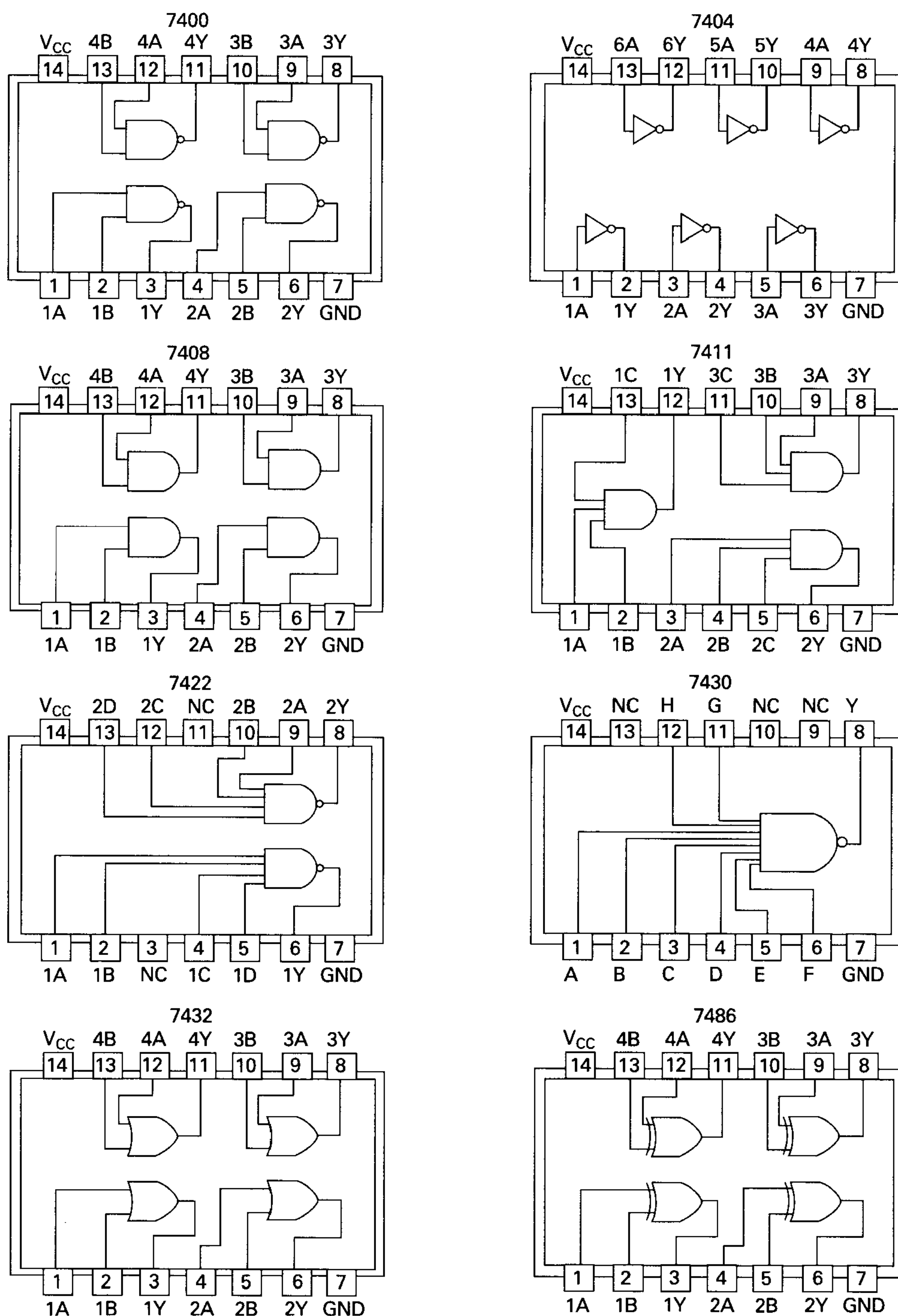
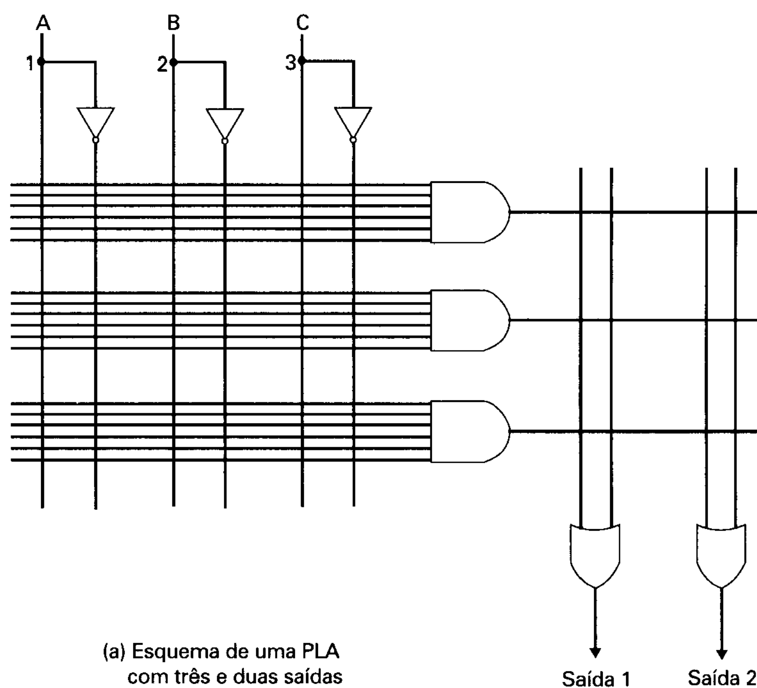
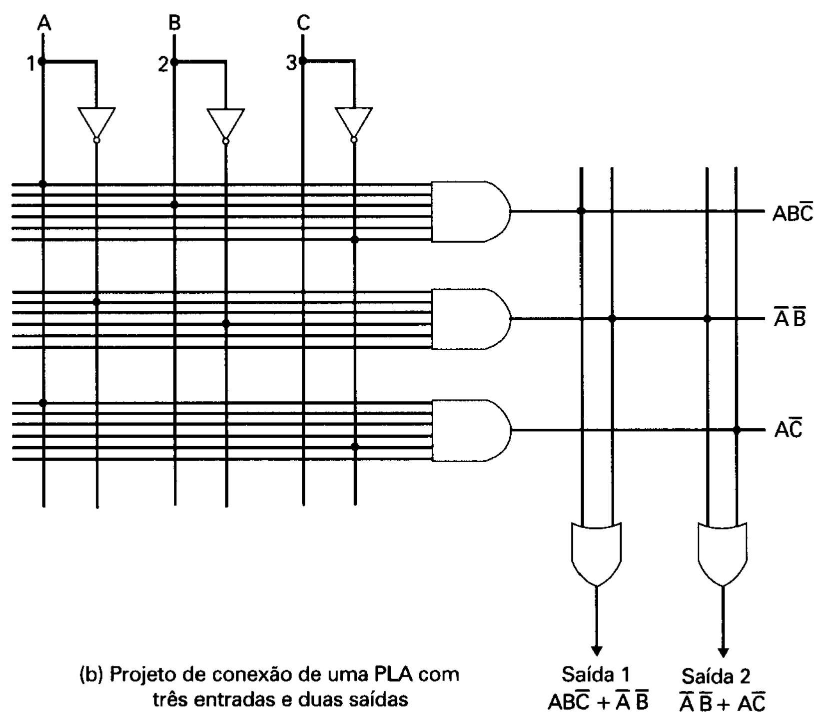


Figura A.18 Algumas pastilhas SSI. Esquemas de pinos extraídos do livro *The TTL Data Book for Design Engineers,* copyright© 1976 by Texas Instrument Incorporated.



(a) Esquema de uma PLA com três e duas saídas



(b) Projeto de conexão de uma PLA com três entradas e duas saídas

Figura A.19 Exemplo de uma matriz de lógica programável.

Memória apenas de leitura

Circuitos combinatórios freqüentemente são chamados de circuitos "sem memória", pois a sua saída depende apenas da entrada corrente, não sendo retido qualquer valor de entradas anteriores. Entretanto, existe um tipo de memória que é implementado usando circuitos combinatórios: a memória apenas de leitura (ROM — *read-only memory*).

Lembre-se de que uma ROM é uma unidade de memória sobre a qual apenas a operação de leitura pode ser efetuada. Isso implica que a informação binária armazenada em uma ROM é permanente e é criada durante o processo de fabricação da ROM. Portanto, uma dada entrada para a ROM (linhas de endereço) sempre produz a mesma saída (linhas de dados). Como as saídas são função apenas das entradas correntes, a ROM é, de fato, um circuito combinatório.

Uma ROM pode ser implementada usando um decodificador e um conjunto de portas OR. Por exemplo, considere a Tabela A.8. Ela pode ser vista como uma tabela verdade com quatro entradas e quatro saídas. Para cada um dos 16 possíveis valores de entrada, é mostrado o valor correspondente de cada linha de saída. Ela também pode ser vista como definindo o conteúdo de uma memória ROM de 64 bits, consistindo de 16 palavras de 4 bits cada. As quatro entradas especificam um endereço, e as quatro saídas especificam o conteúdo da posição de memória especificada pelo endereço. A Figura A.20 mostra como essa memória poderia ser implementada usando um decodificador 4-para-16 e quatro portas lógicas OR. Assim como na PLA, é usada uma organização regular e são feitas conexões que refletem o resultado desejado.

Tabela A.8 Tabela verdade para uma ROM

Entrada				Saída			
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Circuitos somadores

Até agora, vimos como portas lógicas podem ser conectadas para implementar funções tais como roteamento de sinais, decodificação e ROM. Uma área essencial que ainda não foi abordada é a de circuitos para implementar operações aritméticas. Na breve abordagem a seguir, vamos nos contentar em examinar a função de adição.

A adição binária difere da álgebra booleana no fato de que o resultado inclui um termo de 'vai-um' (*carry*). Assim,

$$\begin{array}{r} 0 \\ +0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ +1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ +0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

Entretanto, a adição também pode ser expressa em termos booleanos. Na Tabela A.9a, mostramos a lógica para somar dois bits de entrada e produzir um bit de saída e um bit de 'vai-um'. Essa tabela verdade pode ser facilmente implementada em lógica digital. Entretanto, não estamos interessados apenas em efetuar adição de um único par de bits. Ao contrário, queremos adicionar dois números de n bits. Isso pode ser feito combinando um conjunto de somadores, de modo que o bit de 'vai-um' de um deles seja provido como entrada para o somador seguinte. Um somador de 4 bits é mostrado na Figura A.21.

Para que um somador de múltiplos bits funcione, cada um dos somadores de 1 bit deve ter três entradas, incluindo o bit de 'vai-um' do somador do bit de ordem imediatamente inferior. A Tabela A.9b apresenta uma tabela verdade revisada do somador, considerando três entradas.

As duas saídas podem ser expressas como:

$$\text{Soma} = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{B}\,\overline{C}$$
$$\text{Vai-um} = AB + AC + BC$$

A Figura A.22 é uma implementação usando portas AND, OR e NOT.

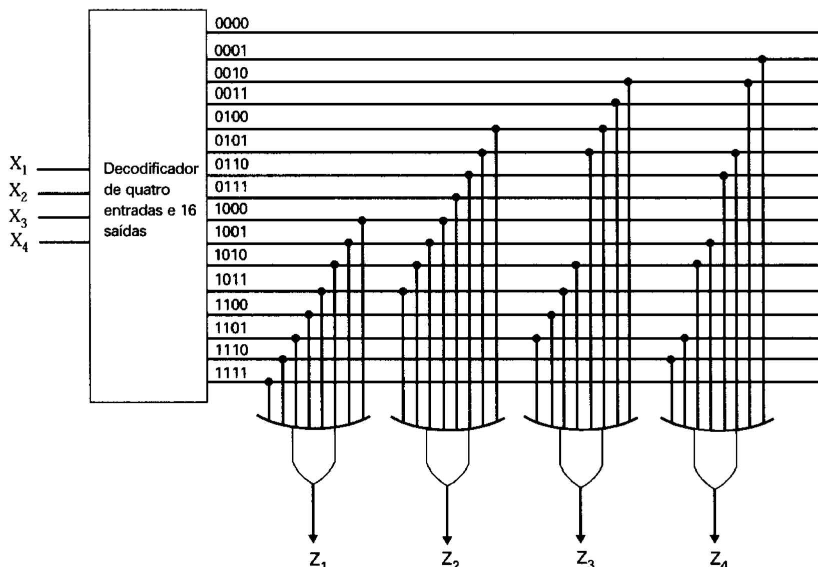


Figura A.20 ROM de 64 bits.

Tabela A.9 Tabelas verdade para a adição binária

(a) Adição de um único bit

A	B	Soma	'Vai-um'
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) Adição com uma entrada de bit de 'vai-um'

C_{in}	A	B	Soma	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

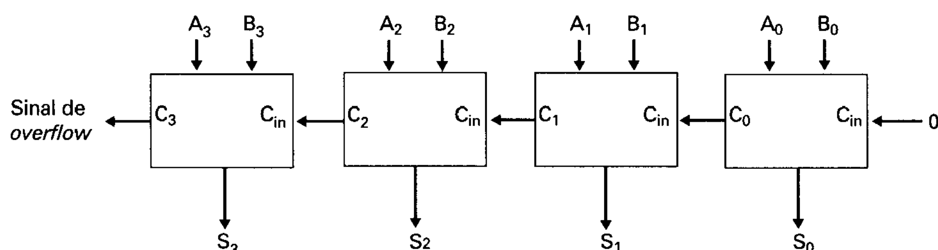


Figura A.21 Somador de 4 bits.

Temos, portanto, a lógica necessária para implementar um somador de múltiplos bits, tal como mostrado na Figura A.23. Note que, como a saída de cada somador depende da saída do sinal de 'vai-um' do somador anterior, há um atraso crescente do bit menos significativo para o bit mais significativo. Cada somador de um bit tem um certo atraso de propagação de sinal através das portas, e esse atraso se acumula. Para somadores muito grandes, esse atraso acumulado torna-se inaceitável.

Se os valores de 'vai-um' puderem ser determinados sem que seja necessário passar por todos os estágios anteriores, cada somador de um bit poderia funcionar independentemente, e o atraso não seria cumulativo. Isso pode ser obtido com uma abordagem conhecida como *'vai-um' antecipado* (*carry lookahead*). Vamos examinar novamente o somador de 4 bits, para explicar essa abordagem.

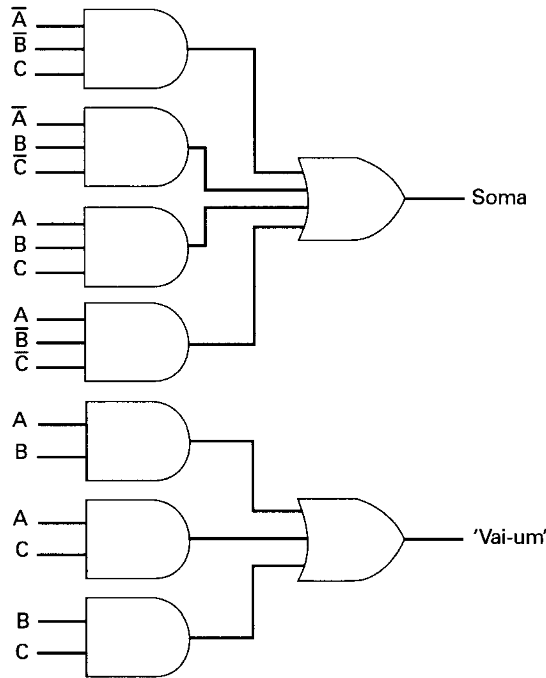


Figura A.22 Implementação de um somador.

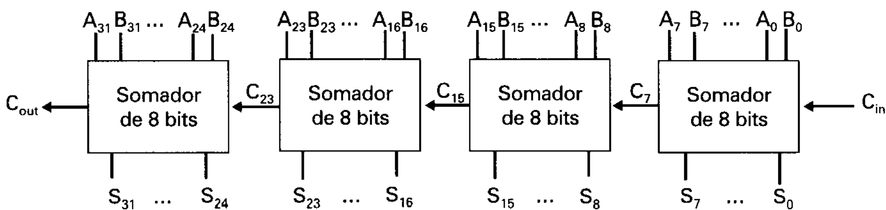


Figura A.23 Construção de um somador de 32 bits usando somador de 8 bits.

Gostaríamos de obter uma expressão que especifica o bit de 'vai-um' na entrada para cada estágio do somador, sem referenciar valores de 'vai-um' de estágios anteriores. Temos:

$$C_0 = A_0B_0 \tag{A.4}$$

$$C_1 = A_1B_1 + A_1A_0B_0 + B_1A_0B_0 \tag{A.5}$$

Seguindo o mesmo procedimento, obtemos:

$$C_2 = A_2B_2 + A_2A_1B_1 + A_2A_1A_0B_0 + A_2B_1A_0B_0 + B_2A_1B_1 + B_2A_1A_0B_0 + B_2B_1A_0B_0$$

Esse processo pode ser repetido para somadores arbitrariamente longos. Cada termo de 'vai-um' pode ser expresso na forma de soma de produtos, como função apenas das entradas originais, não dependendo de qualquer valor de 'vai-um' computado em estágio anterior.

Portanto, ocorrem apenas dois níveis de atraso de propagação de sinal nas portas, independentemente da extensão do somador.

Para números muito grandes, essa abordagem se torna extremamente complicada. Avaliar a expressão para o bit mais significativo de um somador de *n* bits requer uma porta OR com *n*-1 entradas e *n* portas AND, com 2 a *n*+1 entradas. Por isso, um somador completo com 'vai-um' antecipado é tipicamente construído para somar apenas 4 a 8 bits de cada vez. A Figura A.23 mostra como um somador de 32 bits pode ser construído a partir de somadores de 8 bits. Nesse caso, o bit de 'vai-um' tem de ser propagado por meio dos quatro somadores de 8 bits, mas isso é substancialmente mais rápido do que se tivesse de ser propagado por meio de 32 somadores de 1 bit.

A.4 CIRCUITOS SEQÜENCIAIS

Os circuitos combinatórios implementam as funções essenciais de um computador digital. Entretanto, exceto no caso especial da ROM, eles não provêem informação de estado ou memória, elementos também essenciais para a operação de um computador digital. Para esse propósito, é usada uma forma mais complexa de circuito lógico digital: o circuito seqüencial. A saída corrente de um circuito seqüencial depende não apenas da entrada corrente mas também de valores anteriores da entrada. Outra forma, e geralmente mais útil, de ver esses circuitos é que a saída corrente de um circuito seqüencial depende da entrada corrente e do estado do circuito.

Nesta seção, examinamos alguns exemplos simples mas, úteis, de circuitos seqüenciais. Como veremos, um circuito seqüencial faz uso de circuitos combinatórios.

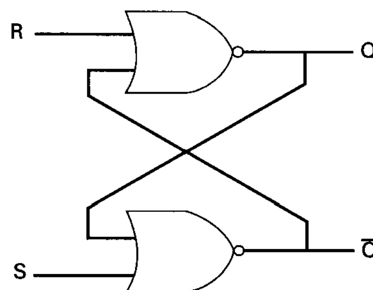


Figura A.24 Flip-flop S–R implementado com portas NOR.

Flip-flops

A forma mais simples de circuito seqüencial é um flip-flop. Existe uma variedade de flip-flops, todos compartilhando duas propriedades:

- O flip-flop é um dispositivo biestável. Ele existe em um de dois estados estáveis e, na ausência de sinal de entrada, permanece nesse estado. Portanto, o flip-flop pode funcionar como uma memória de 1 bit.
- O flip-flop tem duas saídas, que têm sempre valor complementar uma da outra. Essas saídas geralmente são rotuladas como Q e $\overline{Q}$.

Flip-flop S-R

A Figura A.24 mostra uma configuração comum conhecida como flip-flop S–R ou latch S–R. O circuito tem duas entradas, S (Set) e R (Reset), e duas saídas, Q e $\overline{Q}$, e consiste de duas portas NOR, conectadas de forma que a saída retorna como entrada para a outra.

Primeiramente, vejamos por que esse circuito é biestável, ou seja, tem dois estados. Suponha que tanto R quanto S sejam 0 e que Q seja 0. As entradas da porta NOR inferior são Q = 0 e S = 0. Portanto, a saída $\overline{Q}$ = 1 significa que as entradas para a porta NOR superior são $\overline{Q}$ = 1 e R = 0, o que fornece a saída Q = 0. Assim, o estado do circuito é coerente internamente, e permanece estável desde que S = R = 0. Um raciocínio análogo mostra que o estado Q = 1, $\overline{Q}$ = 0 também é estável, para R = S = 0.

Portanto, o circuito pode funcionar como uma memória de 1 bit. Podemos ver a saída Q como o "valor" desse bit. As entradas S e R servem para escrever os valores 1 e 0, respectivamente, na memória. Para ver isso, considere o estado Q = 0, $\overline{Q}$ = 1, S = 0, R = 0. Suponha que o valor de S é alterado para 1. Então, as entradas para a porta NOR inferior serão S = 1 e Q = 0. Depois de um certo atraso Δt, a saída da porta NOR inferior será $\overline{Q}$ = 0 (veja a Figura A.25). Assim, nesse instante, as entradas para a porta NOR superior se tornam R = 0 e $\overline{Q}$ = 0. Depois de outro atraso Δt, a saída Q fica igual a 1. Novamente, esse é um estado estável. As entradas para a porta NOR inferior são agora S = 1, Q = 1, que mantêm a saída $\overline{Q}$ = 0. Desde que tenhamos S = 1 e R = 0, as saídas permanecerão sendo Q = 1 e $\overline{Q}$ = 0. Além disso, se o valor de S é novamente alterado para 0, as saídas permanecem inalteradas.

A entrada R tem a função contrária. Se R for alterado para o valor 1, as saídas serão Q = 0 e $\overline{Q}$ = 1, independentemente do estado anterior de Q e $\overline{Q}$. Novamente, ocorre um atraso igual a 2Δt antes que a estabilidade seja restabelecida (Figura A.25).

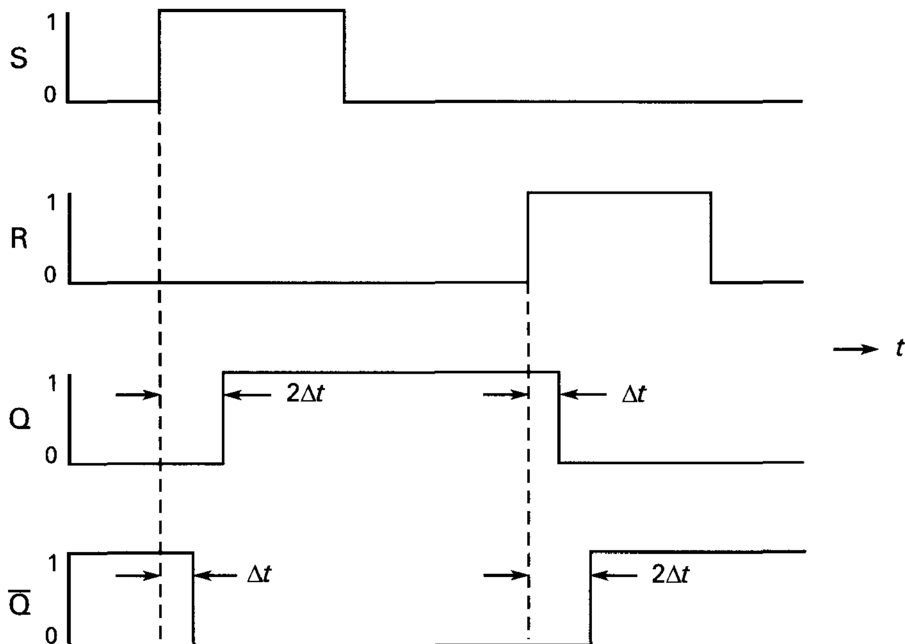


Figura A.25 Diagrama de tempo do flip-flop S–R implementado com portas NOR.

Tabela A.10 O flip-flop S–R

(a) Tabela característica

Entradas correntes	Estado corrente	Próximo estado
SR	Q_n	Q_{n+1}
00	0	0
00	1	1
01	0	0
01	1	0
10	0	1
10	1	1
11	0	—
11	1	—

(b) Tabela característica simplificada

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	—

(c) Resposta para uma série de entradas

t	0	1	2	3	4	5	6	7	8	9
S	1	0	0	0	0	0	0	0	1	0
R	0	0	0	1	0	0	1	0	0	0
Q_{n+1}	1	1	1	0	0	0	0	0	1	1

O flip-flop S–R pode ser definido por uma tabela similar a uma tabela verdade, denominada *tabela característica*, que mostra o próximo estado, ou próximos estados, de um circuito seqüencial, como função do estado e entradas correntes. No caso do flip-flop S–R, o estado pode ser definido como o valor de Q. A Tabela A.10a mostra a tabela característica resultante. Observe que as entradas S = 1, R = 1 não são permitidas, pois produziriam uma saída incoerente (tanto Q quanto $\overline{Q}$ iguais a 0). A tabela pode ser expressa de forma mais compacta tal como na Tabela A.10b. Um exemplo do comportamento do flip-flop S–R é mostrado na Tabela A.10c.

Flip-flop S-R com relógio

A saída do flip-flop S–R muda, depois de um breve atraso, em resposta a uma mudança na entrada. Isso é chamado operação assíncrona. Mais tipicamente, os eventos em um computador digital são sincronizados por um pulso de relógio, de forma que mudanças de sinais só ocorrem quando ocorre um pulso de relógio. A Figura A.26 mostra esse esquema. Esse dispositivo é denominado um flip-flop S–R com relógio (*clocked S-R flip flop*). Note que as entradas S e R passam por meio das portas NOR somente durante o pulso do relógio.

Flip-flop tipo D

Um problema do flip-flop S–R é que a condição R = 1, S = 1 deve ser evitada. Uma forma de fazer isso é permitir apenas uma única entrada, o que é feito pelo flip-flop tipo D. A Figura A.27 mostra a implementação do flip-flop tipo D por meio de portas lógicas e a sua tabela

característica. Usando um inversor, garante-se que as entradas não sincronizadas das duas portas AND tenham valor complementar uma da outra.

O flip-flop tipo D é chamado algumas vezes de flip-flop de dado, porque é, com efeito, uma célula de armazenamento de 1 bit de dado. A saída do flip-flop tipo D é sempre igual ao valor aplicado na entrada mais recentemente. Portanto, ele se lembra e reproduz a última entrada. Ele é também chamado de flip-flop de atraso, porque atrasa de um único pulso de relógio o valor 0 ou 1 que é aplicado na sua entrada.

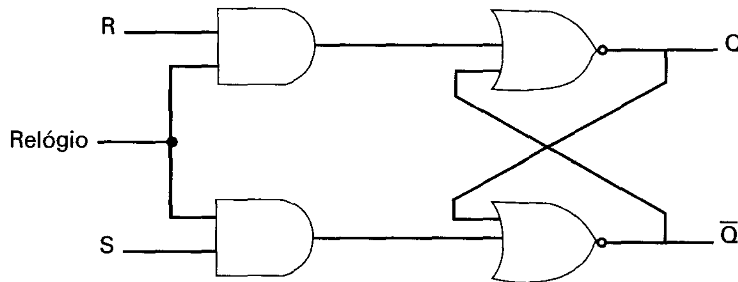


Figura A.26 Flip-flop S-R com relógio.

Flip-flop J-K

Outro flip-flop útil é o J–K. Assim como o flip-flop S–R, ele tem duas entradas. Entretanto, nesse caso, qualquer combinação possível dos valores de entrada é válida. A Figura A.28 mostra uma implementação do flip-flop J–K usando portas lógicas e a Figura A.29 apresenta sua tabela característica (juntamente com as tabelas características dos flip-flops S-R e tipo D). Note que as três primeiras combinações são as mesmas do flip-flop S–R. Se não houver entrada, a saída permanece estável. A entrada J, sozinha, faz com que o valor da saída se torne igual a 1 (set); a entrada K, sozinha, faz com que o valor da saída se torne igual a zero (reset). Quando tanto J quanto K são iguais a 1, o valor da saída é invertido (*toogle*). Assim, se Q é 1 e é aplicado um sinal de valor igual a 1 nas entradas J e K, o valor de Q se torna 0. Você deve verificar que o circuito mostrado na Figura A.28 realmente implementa essa função característica.

D	Q_{n+1}
0	0
1	1

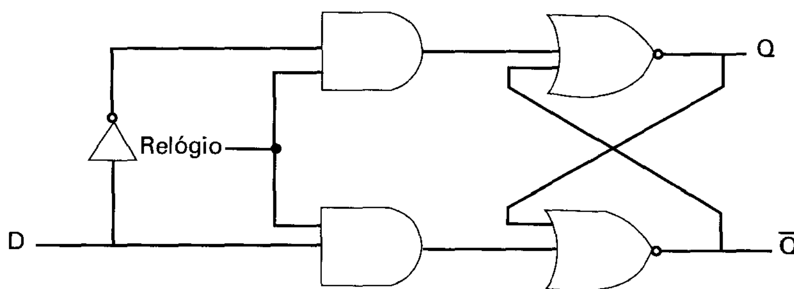


Figura A.27 Flip-flop tipo D.

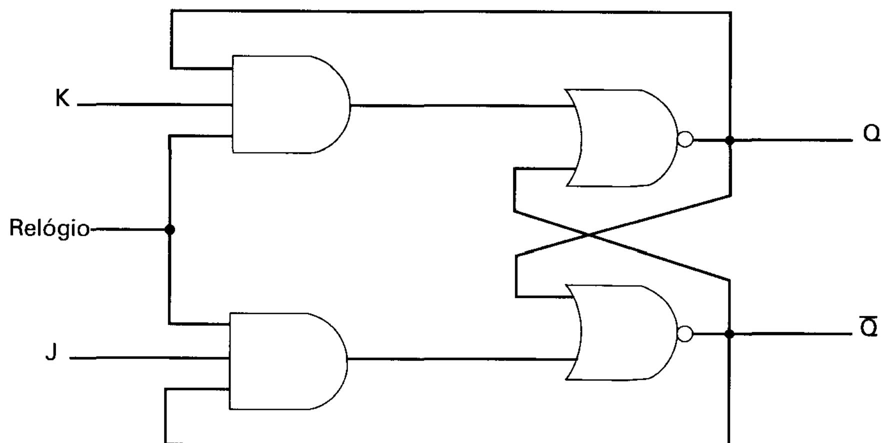


Figura A.28 Flip-flop J–K.

Nome	Símbolo gráfico	Tabela característica
S–R	S Q Ck R $\overline{Q}$	S R Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 –
J–K	J Q Ck K $\overline{Q}$	J K Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 $\overline{Q_n}$
D	D Q Ck $\overline{Q}$	D Q_{n+1} 0 0 1 1

Figura A.29 Flip-flops básicos.

Registradores

Como exemplo de uso de flip-flops, examinemos primeiramente um dos elementos essenciais da CPU: o registrador. Como sabemos, um registrador é um circuito digital usado dentro da CPU para armazenar um ou mais bits de dados. Dois tipos básicos de registradores são comumente utilizados: registradores paralelos e registradores de deslocamento.

Registradores paralelos

Um registrador paralelo consiste de um conjunto de memórias de 1 bit, que podem ser lidas ou escritas simultaneamente. Ele é usado para armazenar dados. Os registradores que discutimos ao longo deste livro são registradores paralelos.

O registrador de 8 bits da Figura A.30 ilustra a operação de um registrador paralelo. Ele é construído usando flip-flops S–R. Um sinal de controle, denominado *habilitação de entrada de dados* (*input data strobe*), controla a escrita no registrador pelas linhas de sinal D11 a D18. Essas linhas podem constituir a saída de um multiplexador, de maneira que dados de uma variedade de fontes possam ser carregados no registrador. A saída é controlada de forma similar. Uma característica extra disponível é a linha usada para atribuir valor zero ao registrador (linha de reset). Note que isso não poderia ser feito facilmente com um registrador construído usando flip-flops tipo D.

Registrador de deslocamento

Um registrador de deslocamento aceita e/ou transfere informação serialmente. Considere, por exemplo, a Figura A.31, que mostra um registrador de deslocamento de 5 bits, construído a partir de flip-flops S–R com relógio. A cada pulso de relógio, os dados são deslocados uma posição para a direita, e o bit mais à direita é transferido para a saída.

Registradores de deslocamento podem ser usados como interface para dispositivos de E/S seriais. Além disso, podem ser usados dentro da ULA, para implementar as funções de deslocamento lógico e rotação. Nesse caso, eles devem também ser equipados com circuitos para leitura/escrita paralela e também serial.

Contadores

Outra categoria de circuitos seqüenciais muito útil é a de contadores. Um contador é um registrador cujo valor é facilmente incrementado de 1 módulo a capacidade do registrador. Um registrador constituído de n flip-flops pode contar até $2^n - 1$. Quando o contador é incrementado além do seu valor máximo, seu valor volta para 0. Um exemplo de contador da CPU é o contador de instruções de programa.

Os contadores podem ser síncronos ou assíncronos, dependendo da sua forma de operação. Contadores assíncronos são relativamente lentos, porque a saída de um flip-flop dispara uma mudança no estado do flip-flop seguinte. Em um contador síncrono, o estado de todos os flip-flops é alterado simultaneamente. Por ser mais rápido, esse tipo de flip-flop é o usado na CPU. Entretanto, é útil começar nossa discussão pela descrição de um contador assíncrono.

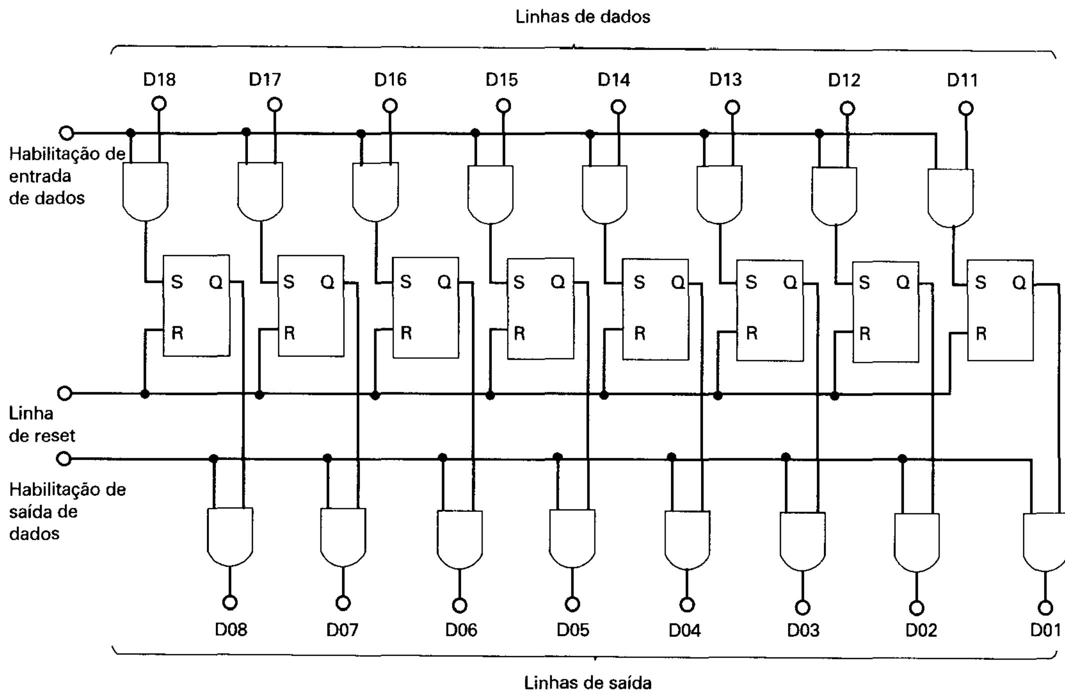


Figura A.30 Registrador paralelo de 8 bits.

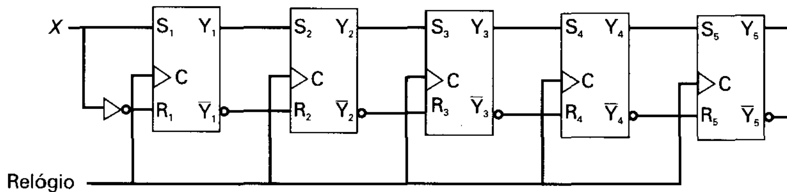


Figura A.31 Registrador de deslocamento de 5 bits.

Em um contador assíncrono, a alteração feita para incrementar o valor do contador começa em uma das suas extremidades e prossegue, passo a passo, até a outra extremidade. A Figura A.32 mostra uma implementação de um contador de 4 bits, usando flip-flops J-K, assim como um diagrama de tempo que ilustra o seu comportamento. O diagrama de tempo é idealizado no sentido de que ele não mostra atrasos de propagação de sinal que ocorrem à medida que o sinal se move ao longo da série de flip-flops. A saída do flip-flop mais à esquerda (Q_0) é o bit menos significativo. Esse projeto pode naturalmente ser estendido para um número arbitrário de bits, encadeando mais flip-flops.

Na implementação mostrada, o contador é incrementado dentro do intervalo de cada pulso de relógio. As entradas J e K de cada flip-flop são mantidas com valor constante igual a 1. Isso significa que, quando ocorre um pulso de relógio, a saída Q é invertida (1 para 0; 0 para 1). Note que a mudança de estado ocorre na borda de descida do pulso de relógio; esse

tipo de flip-flop é conhecido como flip-flop sensível a borda (*edge-triggered flip-flop*). O uso de flip-flops que respondem a uma transição do pulso de relógio, e não ao pulso propriamente dito, provê melhor controle de temporização em circuitos complexos. Se verificarmos os padrões de bits na saída do contador, veremos que a seguinte seqüência de valores é repetida: 0000, 0001, ..., 1110, 1111, 0000, e assim por diante.

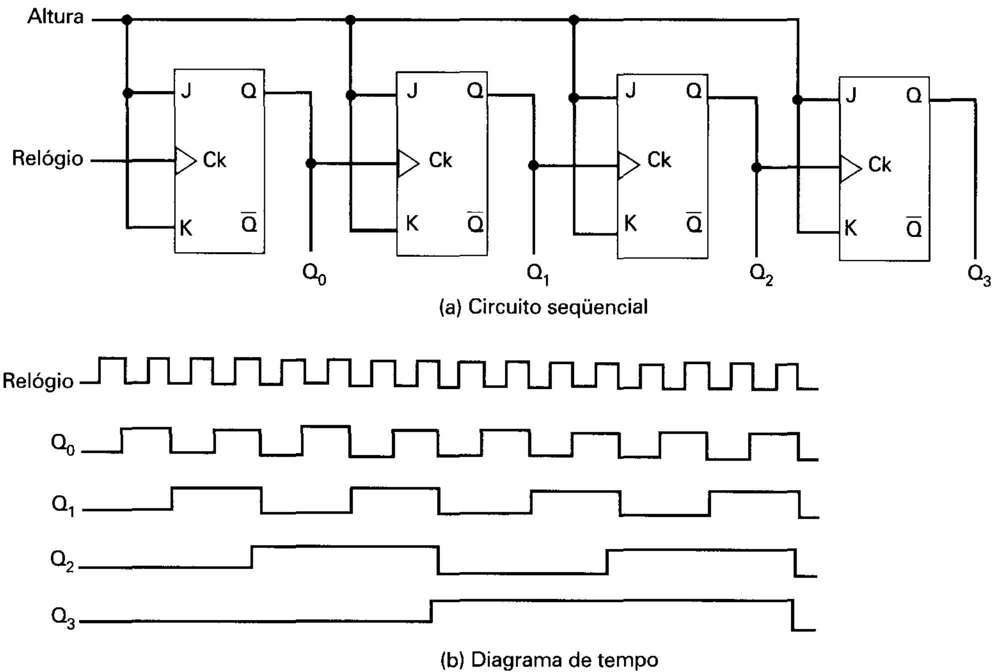


Figura A.32 Contador assíncrono.

Contador assíncrono

O contador assíncrono tem como desvantagem o atraso envolvido na mudança de valor do contador, que é proporcional ao tamanho do contador. Para superar esse problema, a CPU utiliza contadores síncronos, nos quais os estados de todos os flip-flops do contador são alterados ao mesmo tempo. Nessa subseção, apresentamos o projeto de um contador síncrono de 3 bits. Para isso, ilustramos alguns conceitos básicos do projeto de circuitos síncronos.

Para um contador de 3 bits, são requeridos três flip-flops. Usaremos flip-flops J–K. As saídas desses três flip-flops serão denominadas A, B e C, respectivamente, onde C representa o bit menos significativo. O primeiro passo no projeto do circuito do contador síncrono como um todo é a construção da tabela verdade que relaciona as entradas e as saídas dos flip-flops J-K. Essa tabela verdade é mostrada na Figura A.33a. As três primeiras colunas mostram as possíveis combinações das saídas A, B e C. Elas são listadas na ordem em que ocorrem à medida que o contador é incrementado. Cada linha lista o valor corrente de A, B e C e as entradas para os três flip-flops requeridas para obter o próximo valor de A, B e C.

Para entender como a tabela verdade da Figura A.33a é construída, pode ser útil rever a tabela característica do flip-flop J–K. Essa tabela é a seguinte:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

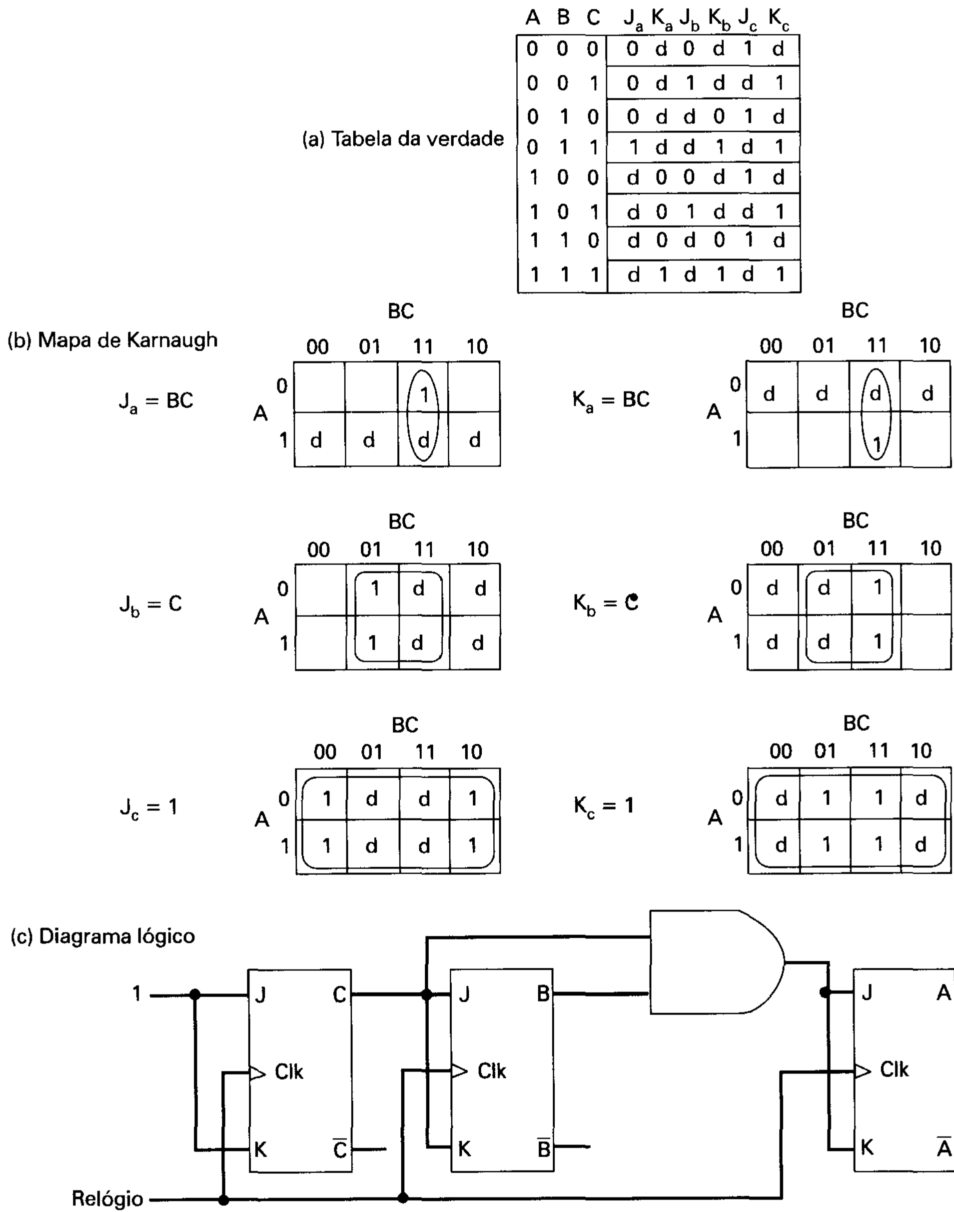
Nessa forma, a tabela mostra o efeito das entradas J e K sobre a saída. Considere agora a seguinte organização da mesma informação:

Q_n	J	K	Q_{n+1}
0	0	d	0
0	1	d	1
1	d	1	0
1	d	0	1

Nessa forma, a tabela provê o valor da próxima saída, quando são conhecidas as entradas e a saída corrente. Essa é exatamente a informação necessária para projetar o contador, ou, de fato, qualquer circuito seqüencial. Uma tabela nessa forma é conhecida como *tabela de excitação*.

Voltemos à Figura A.33a. Considere a primeira linha. Queremos que o valor de A permaneça igual a 0, o valor de B permaneça igual a 0 e o valor de C passe de 0 para 1, durante a aplicação do próximo pulso de relógio. A tabela de excitação mostra que para manter uma saída com valor 0, devemos ter J = 0 e qualquer valor na entrada K (K = d). Esses valores são mostrados na primeira linha da tabela. Usando um raciocínio similar, é possível preencher o restante da tabela.

Uma vez construída a tabela da Figura A.33a, vemos que essa tabela mostra os valores requeridos para as entradas J e K em função dos valores correntes nas saídas A, B e C. Com o auxílio de mapas de Karnaugh, podemos obter uma expressão booleana para cada uma dessas seis funções. Isso é mostrado na parte (b) da figura. Por exemplo, o mapa de Karnaugh para a variável J_a (a entrada J do flip-flop que produz a saída A) resulta na expressão $J_a = BC$. Tendo derivado as seis expressões, o projeto do circuito é obtido diretamente, como se mostra na parte (c) da figura.



(a) Tabela da verdade

A	B	C	J_a	K_a	J_b	K_b	J_c	K_c
0	0	0	0	d	0	d	1	d
0	0	1	0	d	1	d	d	1
0	1	0	0	d	d	0	1	d
0	1	1	1	d	d	1	d	1
1	0	0	d	0	0	d	1	d
1	0	1	d	0	1	d	d	1
1	1	0	d	0	d	0	1	d
1	1	1	d	1	d	1	d	1

Figura A.33 Projeto de um contador síncrono.

A.5 EXERCÍCIOS

A.1 Construa a tabela verdade para as seguintes expressões booleanas:

a. ABC + $\overline{A}\,\overline{B}\,\overline{C}$

b. ABC + A$\overline{B}\,\overline{C}$ + $\overline{A}\,\overline{B}\,\overline{C}$

c. A(B$\overline{C}$+$\overline{B}$C)

d. (A + B)(A + C)($\overline{A}$ + $\overline{B}$)

A.2 Simplifique as seguintes expressões, usando as leis de comutatividade:

a. $A \bullet \overline{B} + \overline{B} \bullet A + C \bullet D \bullet E + \overline{C} \bullet D \bullet E + E \bullet \overline{C} \bullet D$

b. $A \bullet B + A \bullet C + B \bullet A$

c. $(L \bullet M \bullet N)(A \bullet B)(C \bullet D \bullet E)(M \bullet N \bullet L)$

d. $F \bullet (K + R) + S \bullet V + W \bullet \overline{X} + V \bullet S + \overline{X} \bullet W + (R + K) \bullet F$

A.3 Aplique as leis de DeMorgan às seguintes expressões:

a. $F = \overline{V + A + L}$

b. $F = \overline{A} + \overline{B} + \overline{C} + \overline{D}$

A.4 Simplifique as seguintes expressões:

a. $A = S \bullet T + V \bullet W + R \bullet S \bullet T$

b. $A = T \bullet U \bullet V + X \bullet Y + Y$

c. $A = F \bullet (E + F + G)$

d. $A = (P \bullet Q + R + S \bullet T)T \bullet S$

e. $A = \overline{\overline{D} \bullet \overline{D} \bullet E}$

f. $A = Y \bullet (W + X + \overline{\overline{Y} + \overline{Z}}) \bullet Z$

g. $A = (B \bullet E + C + F) \bullet C$

A.5 Construa a operação XOR a partir das operações booleanas básicas AND, OR e NOT.

A.6 Dada uma porta NOR e portas NOT, desenhe um diagrama lógico que implemente a função AND sobre três entradas.

A.7 Escreva uma expressão booleana para uma porta NAND de quatro entradas.

A.8 Um circuito combinatório é usado para controlar um mostrador digital de sete segmentos, usado para exibir números decimais, tal como mostrado na Figura A.34. O circuito tem quatro entradas, que usam um código de 4 bits como na representação de número decimal empacotado BCD (0_{10} = 0000, ..., 9_{10} = 1001). As sete saídas definem que segmentos do mostrador devem ser ativados para exibir cada número decimal. Note que algumas combinações das entradas e das saídas não são permitidas.

a. Construa a tabela verdade para esse circuito.

b. Expresse essa tabela verdade na forma de soma de produtos (SOP).

c. Expresse essa tabela verdade na forma de produto de somas (POS).

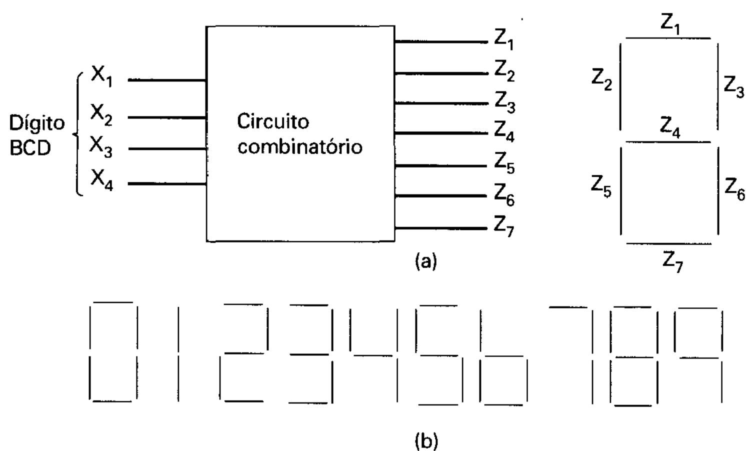


Figura A.34 Exemplo de mostrador digital com sete segmentos.

A.9 Projete um multiplexador 8-para-1.

A.10 Inclua uma linha adicional na Figura A.15, de modo que o circuito funcione como um demultiplexador.

A.11 O Código de Gray é um código binário para números inteiros. Ele difere da representação binária normal, no fato de que existe alteração de apenas um bit da representação de um número para a representação do número consecutivo. Essa representação é útil para aplicações onde é gerada uma seqüência de números, tais como em contadores ou conversores analógico-digitais. Como apenas um bit é alterado a cada instante, nunca ocorrerão ambigüidades devidas a pequenas diferenças de temporização de sinais. Os oito primeiros elementos do código de Gray são:

Código binário	Código de gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Projete um circuito que converte do código binário para o código de Gray.

A.12 Projete um decodificador 5 × 32, usando quatro decodificadores 3 × 8 (com entrada de habilitação) e um decodificador 2 × 4.

A.13 Implemente o somador da Figura A.22 usando apenas cinco portas (*Dica*: use algumas portas XOR).

A.14 Considere a Figura A.22. Suponha que cada porta produz um atraso de sinal de 10 ns. Portanto, a saída da soma é válida somente 30 ns depois e a saída de 'vai-um' 20 ns depois. Qual seria o tempo gasto por um somador de 32 bits para efetuar uma adição, em cada um dos casos a seguir:

a. O somador é implementado sem 'vai-um' antecipado, tal como na Figura A.21.

b. O somador é implementado com 'vai-um' antecipado, usando somadores de 8 bits, tal como na Figura A.23.

apêndice

B

PROJETOS PARA O ENSINO DE ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Muitos professores acreditam que projetos de pesquisa e de implementação são cruciais para um bom entendimento dos conceitos de arquitetura e organização de computadores. Sem tais projetos, pode ser difícil para o estudante adquirir experiência sobre alguns conceitos básicos e as interações entre eles. Projetos reforçam os conceitos introduzidos pelo livro e possibilitam ao estudante apreciar melhor o funcionamento interno dos processadores, além de motivá-lo e dar-lhe a confiança de que dominam os assuntos estudados.

Neste texto, procuramos apresentar os conceitos tão claramente quanto possível e fornecer quase 200 exercícios para reforçar a compreensão desses conceitos. Muitos professores gostarão de poder complementar esse material com projetos. Este apêndice oferece algumas sugestões nesse sentido e descreve o material de suporte adicional contido no manual do professor. Esse material de suporte cobre três tipos de projeto:

- Projetos de pesquisa
- Projetos de simulação
- Atividades de leitura e relatório

B.1 PROJETOS DE PESQUISA

Uma forma eficaz de reforçar os conceitos básicos do curso e orientar o estudante sobre atividades de pesquisa é desenvolver um projeto de pesquisa. Tal projeto pode envolver uma revisão da literatura existente, assim como a pesquisa na WEB sobre produtos comerciais, atividades de laboratórios de pesquisa e esforços de padronização. Os projetos devem ser feitos em grupo ou, no caso de projetos menores, individualmente. Em qualquer dos casos, é melhor que o projeto seja proposto no início do período letivo, dando tempo ao professor para avaliar a adequação do projeto em termos do tema proposto e do nível de esforço requerido. Instruções para que o aluno desenvolva o projeto de pesquisa devem incluir:

- O formato da proposta de projeto
- O formato do relatório final do projeto
- Um cronograma incluindo resultados de etapas intermediárias e prazos finais
- Uma lista de possíveis tópicos de projeto

Os estudantes poderão selecionar um dos tópicos listados ou propor um projeto comparável. O manual do professor inclui uma sugestão de formato para a proposta e para o relatório final, assim como uma lista de possíveis tópicos de pesquisa.

B.2 PROJETOS DE SIMULAÇÃO

Uma excelente forma para que o estudante possa obter experiência sobre a operação interna de um processador e estudar e apreciar algumas das questões de projeto e suas implicações sobre o desempenho é pela simulação de elementos-chave do processador. Uma ferramenta útil para isso é o SimpleScalar.

Tanto na pesquisa como para fins educacionais, a simulação oferece diversas vantagens sobre a implementação efetiva do hardware:

- Com a simulação, é mais fácil modificar os vários elementos da organização, para alterar características de desempenho dos diversos componentes, e então analisar os efeitos dessas modificações.
- A simulação fornece diversas estatísticas detalhadas de desempenho, que podem ser usadas para entender a influência de características dos vários componentes sobre o desempenho global do sistema.

O SimpleScalar (Burger e Austin, 1997) é um conjunto de ferramentas usadas para simular problemas reais relativos a sistemas e processadores modernos. Esse conjunto de ferramentas inclui um compilador, um montador, um ligador e ferramentas de simulação e visualização. O SimpleScalar fornece simuladores de processador, que variam desde um simulador de características funcionais, extremamente rápido, até um simulador de processador superescalar com execução fora de ordem, em grande nível de detalhe, que provê suporte para caches não bloqueáveis e execução especulativa. A arquitetura do conjunto de instruções e os parâmetros de organização podem ser modificados para criar uma variedade de experimentos.

O manual do professor deste livro inclui uma introdução sucinta sobre o SimpleScalar para os estudantes, com instruções sobre como instalar e usar o SimpleScalar. O manual inclui também algumas sugestões de projeto de simulação.

O SimpleScalar é um software portável, que pode ser executado no Windows NT e na maioria das plataformas UNIX. O SimpleScalar pode ser obtido na página Web do SimpleScalar, sem qualquer ônus para uso não-comercial.

B.3 ATIVIDADES DE LEITURA E RELATÓRIO

Outra excelente forma de reforçar os conceitos do curso e prover ao estudante alguma experiência de pesquisa é selecionar artigos técnico-científicos para serem lidos e analisados pelos estudantes. O manual do professor inclui uma lista de artigos sugeridos, um ou dois por capítulo. Todos os artigos podem ser facilmente obtidos pela Internet ou em uma boa biblioteca universitária. O manual inclui também uma sugestão de formato de relatório de projeto desse tipo.

GLOSSÁRIO

A definição de alguns dos termos deste glossário foi extraída do *ANSI — American National Dictionary for Information Systems* (1990). Eles estão indicados aqui com um asterisco.

Acesso Direto (Direct Access)* Capacidade de obter dados de um dispositivo de memória ou de armazenar dados nesse dispositivo de memória de forma independente da posição relativa dos dados, com base em um endereço que indica a posição física do dado.

Acesso Direto à Memória (DMA — Direct Memory Access) Forma de E/S em que um módulo especial, denominado módulo de DMA, controla a transferência de dados entre a memória principal e um dispositivo de E/S. A CPU envia ao módulo de DMA uma requisição de transferência de um bloco de dados e é interrompida apenas quando todo o bloco for transferido.

Acumulador (Accumulator) Registrador usado em uma CPU com formato de instruções de um único endereço. O acumulador, ou AC, é implicitamente um dos dois operandos da instrução.

Arbitração do Barramento (Bus Arbitration) Processo usado para determinar, entre um conjunto de mestres que competem pelo uso do barramento, qual deles deterá o controle para usar o barramento.

Arranjo Lógico Programável (PLA — Programmable Logic Array)* Arranjo de portas lógicas cujas interconexões podem ser programadas para implementar uma função lógica específica.

Arranjo Redundante de Discos Independentes (RAID — Redundant Array of Independent Disks) Arranjo de discos no qual parte da capacidade física de armazenamento é usada para armazenar informação redundante sobre os dados armazenados no restante da capacidade de armazenamento. A informação redundante possibilita a regeneração de dados, em caso de falha em um dos elementos do arranjo de discos ou em um dos caminhos de dados.

ASCII (American Standard Code for Information Interchange) Código padrão americano para troca de informações. O ASCII é um código de 7 bits usado para representar caracteres numéricos, caracteres alfabéticos e caracteres imprimíveis especiais. Ele inclui ainda *caracteres de controle*, que não podem ser impressos ou exibidos na tela, e são usados para funções de controle.

Auto-indexação (Autoindexing) Forma de endereçamento por indexação em que o registrador índice é incrementado ou decrementado automaticamente, quando é usado em uma referência à memória.

Barramento (Bus) Caminho de comunicação compartilhado, constituído por um conjunto de linhas. Em alguns sistemas de computação, a CPU, a memória e os dispositivos de E/S

são conectados a um barramento comum. Como as linhas são compartilhadas entre todos os componentes, apenas um deles pode, em cada instante, efetivamente transmitir dados ou sinais.

Barramento de Controle (Control Bus) Parte do barramento de sistema usada para transferência de sinais de controle.

Barramento de Dados (Data Bus) Porção do barramento de sistema usada para transferência de dados.

Barramento de Endereço (Address Bus) É a parte do barramento do sistema usada para transferir um endereço. Tipicamente, o endereço transferido identifica uma posição de memória ou um dispositivo de E/S.

Barramento de Sistema (System Bus) Barramento usado para conectar os componentes principais do computador (CPU, memória, E/S).

Base (Base)* Nos sistemas de numeração comumente usados em artigos científicos, é o número que é elevado à potência especificada pelo expoente e então multiplicado pela mantissa para determinar o valor do número real representado (por exemplo, o número 6,25 na expressão $2{,}7 \times 6{,}25^{1{,}5} = 42{,}1875$).

Bit (Bit — Binary Digit) * Dígito binário. Dígitos 0 ou 1 do sistema de numeração binário.

Bit de Paridade (Parity Bit)* Dígito binário adicionado a um grupo de dígitos binários para tornar a soma de todos os dígitos ou sempre par (paridade par) ou sempre ímpar (paridade ímpar).

Bloco de Controle de Processo (Process Control Block) Manifestação de um processo em um sistema operacional. É a estrutura de dados que contém informações sobre as características e o estado de um processo.

Bloco de Memória (Page Frame)* No contexto de sistemas de paginação, área da memória principal usada para carregar uma página.

Buffer ou Área de Armazenamento Temporário (Buffer)* Área de memória usada para armazenar dados temporariamente, para compensar diferenças na taxa do fluxo de dados ou na ocorrência de eventos na transferência de dados de um dispositivo para outro.

Byte (Byte) Seqüência de 8 bits. Também conhecido como *octeto*.

Cadeia Circular (Daisy Chain)* Método de conexão de dispositivos usado para determinar prioridade de interrupção, no qual as fontes de interrupção são conectadas de forma serial.

Canal de E/S (I/O Channel) Módulo de E/S relativamente complexo, que libera a CPU de detalhes de operações de E/S. Um canal de E/S executa uma seqüência de comandos de E/S armazenada na memória principal sem requerer o envolvimento da CPU.

Canal Multiplexador (Multiplexor Channel) Canal projetado para operar simultaneamente com diversos dispositivos de E/S. Diversos dispositivos de E/S podem transferir registros ao mesmo tempo, pela intercalação dos dados dos diferentes dispositivos. Veja também *canal multiplexador de byte* e *canal multiplexador de bloco.*

Canal Multiplexador de Bloco (Block Multiplexor Channel) Um canal multiplexador que intercala blocos de dados. Veja também *canal multiplexador de byte.* Contrapõe-se a *canal seletor.*

Canal Multiplexador de Byte (Byte Multiplexor Channel)* Canal multiplexador que intercala bytes de dados. Veja também *canal multiplexador de bloco.* Contrapõe-se a *canal seletor.*

Canal Seletor (Selector Channel) Canal de E/S projetado para operar com apenas um dispositivo de E/S em cada instante. Uma vez que é selecionado um dispositivo de E/S, um

registro completo é transferido, um byte de cada vez. Contrapõe-se a *canal multiplexador de byte* ou *canal multiplexador de bloco.*

CD-ROM (CD-ROM — Compact Disk Read-Only Memory) Memória de apenas leitura em disco compacto. Disco não-apagável usado para armazenar dados no computador. O sistema Padrão usa discos de 12 cm, que podem armazenar mais de 550 Mbytes.

Ciclo de Busca (Fetch Cycle) Porção do ciclo de instrução na qual a CPU busca na memória a próxima instrução a ser executada.

Ciclo de Execução (Execute Cycle) Porção do ciclo de instrução na qual a CPU efetua a operação especificada pelo código de operação da instrução.

Ciclo de Indireção (Indirect Cycle) Porção do ciclo de instrução durante o qual a CPU efetua acesso à memória para converter um endereço indireto em um endereço direto.

Ciclo de Instrução (Instruction Cycle) Processamento efetuado pela CPU para executar uma única instrução.

Ciclo de Interrupção (Interrupt Cycle) Porção do ciclo de instrução durante o qual a CPU verifica se existem interrupções pendentes. Se existir uma interrupção habilitada pendente, a CPU salva o estado do programa corrente e executa uma rotina de tratamento de interrupção.

Circuito Combinatório (Combinational Circuit)* Dispositivo lógico cujos valores de saída, em cada instante, dependem apenas dos valores de entrada nesse instante. Um circuito combinatório é um caso especial de circuito seqüencial, que não possui capacidade de memória.

Circuito Integrado (CI, Integrated Circuit — IC) Pequena peça de material sólido, tal como silício, na qual é estampada uma coleção de componentes eletrônicos e conexões entre eles.

Circuito Seqüencial (Sequential Circuit) Circuito lógico digital cuja saída depende da entrada corrente e do estado do circuito. Circuitos seqüenciais possuem, portanto, o atributo de memória.

Cluster ou Agregado de Computadores (Cluster) Grupo de computadores completos interconectados como um sistema de computação unificado, criando a ilusão de ser uma única máquina. O termo *computador completo* significa um sistema capaz de ser executado independentemente, fora do cluster.

Código de Condição (Condition Code) Código que reflete o resultado de uma operação executada previamente (por exemplo, uma instrução aritmética). Uma CPU pode incluir um ou mais códigos de condição, que podem ser armazenados separadamente dentro da CPU ou como parte de um registrador de controle. Também conhecido como *flag*.

Código de Correção de Erro (Error-Correcting Code)* Código no qual cada caractere ou sinal obedece a regras de construção específicas, de modo que um desvio dessas regras indica presença de um erro e no qual alguns ou todos os erros detectados podem ser corrigidos automaticamente.

Código de Detecção de Erro (Error-Detecting Code)* Código no qual cada caractere ou sinal obedece a regras de construção específicas, de modo que um desvio dessas regras indica presença de um erro.

Código de Operação (Operation Code)* Código usado para representar operações de um computador.

Componente de Estado Sólido (Solid-State Component)* Componente cuja operação depende do controle de fenômeno elétrico ou magnético em sólidos (por exemplo, diodo transistor de cristal, núcleo de ferrite).

Comunicação de Dados (Data Communication) Transferência de dados entre dispositivos. O termo geralmente exclui transferências de E/S.

Conjunto de Instruções de Computador (Computer Instruction Set)* Conjunto completo de operadores de instruções de um computador, juntamente com a descrição dos tipos de significados que podem ser atribuídos a seus operandos. Sinônimo de *conjunto de instruções de máquina.*

Contador de Programa (Program Counter) Registrador de endereço de instrução que armazena o endereço da próxima instrução a ser executada.

Controlador de E/S (I/O Controller) Módulo de E/S relativamente simples, que requer controle detalhado da CPU ou de um canal de E/S. Sinônimo de *controlador de dispositivo.*

CPU Microprogramada (Microprogrammed CPU) CPU cujo controle é implementado por microprogramação.

Decodificador (Decoder)* Dispositivo que possui um certo número de linhas de entrada, das quais algumas podem carregar sinais, e um certo número de linhas de saída, das quais no máximo uma carrega um sinal, existindo uma correspondência biunívoca entre as saídas e as combinações dos sinais de entrada.

Desvio Condicional (Conditional Jump)* Desvio que ocorre apenas quando uma instrução que especifica o desvio é executada e as condições especificadas para o desvio são satisfeitas. Contrapõe-se a *desvio incondicional.*

Desvio Incondicional (Unconditional Jump)* Desvio que ocorre sempre que a instrução que o especifica é executada.

Disco Compacto (CD — Compact Disk) Disco não-apagável que armazena informação de áudio digitalizada.

Disco Magnético (Magnetic Disk)* Prato circular plano com superfície magnetizada, em um ou ambos os lados, no qual podem ser armazenados dados.

Disco Óptico Apagável (Erasable Optical Disk) Disco que emprega tecnologia óptica, mas pode ser facilmente apagado e reescrito. Existem em uso discos de 3,25 e 5,25 polegadas. A capacidade típica é de 650 Mbytes.

Disquete (Diskette)* Disco magnético flexível, em embalagem protetora, usado como memória de segurança para dados de computador. Sinônimo de *disco flexível.*

E/S Dirigida por Interrupção (Interrupt-Driven I/O) Uma forma de E/S. A CPU emite um comando de E/S, continua a executar as instruções subseqüentes e é interrompida pelo módulo de E/S quando este último completa seu trabalho.

E/S Independente (Isolated I/O) Método de endereçamento de módulos de E/S e dispositivos externos. O espaço de endereçamento de E/S é tratado separadamente do espaço de endereçamento da memória principal. Instruções específicas de E/S devem ser usadas. Compare com *E/S mapeada na memória.*

E/S Mapeada na Memória (Memory-Mapped I/O) Método de endereçamento de módulos de E/S e dispositivos externos. Um único espaço de endereçamento é usado tanto para a memória principal como para endereços de E/S, e as mesmas instruções de máquina são usadas para leitura e escrita na memória e para E/S.

E/S Programada (Programmed I/O) Forma de E/S na qual a CPU envia um comando de E/S para um módulo de E/S e tem de esperar que a operação seja completada antes de prosseguir a execução.

Emulação (Emulation)* Imitação de todo ou parte de um sistema por outro sistema, primariamente por hardware, de modo que o sistema imitador recebe os mesmos dados, executa os mesmos programas e obtém os mesmos resultados que o sistema imitado.

Endereço Absoluto (Absolute Address)* Um endereço em uma linguagem de computação que identifica uma posição de memória ou um dispositivo, sem o uso de nenhuma referência intermediária.

Endereço Base (Base Address)* Valor numérico usado como referência no cálculo de um endereço durante a execução de um programa.

Endereço Direto (Direct Address)* Endereço que designa uma posição de memória de um item de dado a ser tratado como operando. Sinônimo de *endereço de nível um* (one-level address).

Endereço Imediato (Immediate Address)* Conteúdo de um campo de endereço que contém o valor de um operando, e não o endereço do operando. Sinônimo de *endereço de nível 0*.

Endereço Indexado (Indexed Address)* Endereço que é modificado pelo valor de um registrador índice, antes ou depois da execução de uma instrução de computador.

Endereço Indireto (Indirect Address)* Endereço de uma posição de memória que contém um endereço.

Entrada-Saída (E/S, Input-Output — I/O) Relativo a entrada ou saída, ou a ambos. Refere-se à movimentação de dados entre o computador e um dispositivo periférico a ele conectado.

Equipamento Periférico (IBM — Peripheral Equipment) Em um sistema de computador, equipamento que provê comunicação com o mundo exterior a uma unidade de processamento particular. Sinônimo de *dispositivo periférico*.

Escalar (Scalar)* Quantidade caracterizada por um único valor.

Espaço de Endereçamento (Address Space) O intervalo de endereços (memória e E/S) que podem ser usados.

Execução Especulativa (Speculative Execution) Execução de instruções ao longo do caminho de um desvio. Se mais tarde se verifica que esse desvio não foi tomado, os resultados da execução especulativa são descartados.

Execução Predicativa (Predicated Execution) Mecanismo que provê suporte à execução condicional de instruções individuais. Isso torna possível a execução especulativa dos dois ramos de uma instrução de desvio e a conservação dos resultados do ramo do desvio que foi tomado em última instância.

Falta de Página (Page Fault) Ocorre quando a página que contém uma palavra referenciada não está na memória principal. Isso causa uma interrupção e requer que o sistema operacional traga a página necessária para a memória principal.

Firmware (Firmware)* Microcódigo armazenado em memória apenas de leitura.

Fita Magnética (Magnetic Tape)* Fita com superfície magnetizada, na qual podem ser armazenados dados por gravação magnética.

Flip-Flop (Flip-Flop)* Circuito ou dispositivo que contém elementos ativos com dois possíveis estados estáveis, podendo estar em um único estado em cada instante. Sinônimo de *circuito biestável*.

Formato de Instrução (Instruction Format) Leiaute de uma instrução de computador como uma seqüência de bits. O formato divide a instrução em campos, correspondendo aos elementos constituintes da instrução (por exemplo, código de operação, operandos etc.).

G Prefixo que significa *bilhão*.

Indexação (Indexing) Técnica de endereçamento em que o endereço é modificado pelo valor de um registrador índice.

Indireção ou ciclo indireto (Indirect) Passo da execução de uma instrução em que é efetuada a busca de um operando com endereçamento indireto.

Instrução de Computador (Computer Instruction)* Instrução que pode ser reconhecida pela unidade de processamento do computador para a qual ela é designada. Sinônimo de *instrução de máquina*.

Interrupção (Interrupt)* Suspensão de um processo, tal como a execução de um programa de computador, causada por um evento externo ao processo, e feita de forma que a execução do processo possa ser retomada.

Interrupção Desabilitada (Disabled Interrupt) Condição, usualmente criada pela CPU, durante a qual a CPU ignora qualquer sinal de requisição de interrupção de uma determinada classe.

Interrupção Habilitada (Enabled Interrupt) Condição, usualmente criada pela CPU, durante a qual a CPU responde a qualquer sinal de requisição de interrupção de uma classe especificada.

K Prefixo que significa $2^{10} = 1024$. Assim, 2 Kb = 2048 bits.

Linguagem de Microprogramação (Microprogramming Language) Conjunto de instruções para especificar microprogramas.

Linguagem de Montagem (Assembly Language)* Linguagem de programação cujas instruções estão em correspondência biunívoca com as instruções de um computador e podem prover recursos como a possibilidade de definição de *macros* (ou macroinstruções). É sinônimo de *linguagem dependente de computador*.

Linha de Cache (Cache Line) Bloco de dados associado a um rótulo na cache e a unidade de transferência de dados entre a cache e a memória.

Localidade de Referência (Locality of Reference) Tendência do processador de referenciar repetidamente o mesmo conjunto de posições de memória durante um curto período de tempo.

M Prefixo que significa $2^{20} = 1.048.576$. Assim, 2 Mb = 2×2^{20} bits.

Memória Associativa (Associative Memory)* Memória onde cada posição de armazenamento de dados é identificada pelo seu conteúdo, ou parte do seu conteúdo, e não por um valor que identifica sua posição.

Memória Cache (Cache Memory)* Memória especial de armazenamento temporário, menor e mais rápida que a memória principal, usada para armazenar uma cópia de instruções ou dados da memória principal mais prováveis de ser requeridos pelo processador em um futuro próximo; essas instruções e dados são obtidos automaticamente da memória principal.

Memória de Acesso Aleatório (RAM — Random-Access Memory) Memória na qual cada posição endereçável tem um mecanismo de endereçamento distinto. O tempo de acesso a uma dada posição é independente da seqüência de acessos anteriores.

Memória de Apenas Leitura (ROM — Read-Only Memory) Memória semicondutora cujo conteúdo não pode ser alterado, exceto pela destruição da unidade de memória. Memória não-apagável.

Memória de Apenas Leitura Programável (PROM — Programmable Read-Only Memory) Memória semicondutora cujo conteúdo pode ser escrito apenas uma vez. O processo de escrita é feito eletricamente e pode ser efetuado pelo usuário após a fabricação original da pastilha.

Memória de Controle (Control Storage) Parte da memória que contém o microcódigo das instruções.

Memória Não-Volátil (Nonvolatile Memory) Memória cujo conteúdo é estável e não requer suprimento de energia constante.

Memória Principal (Main Memory)* Memória endereçável por programa, a partir da qual dados e instruções podem ser diretamente carregados em registradores, para subseqüente processamento ou execução.

Memória Secundária (Secondary Memory) Memória localizada fora do próprio sistema de computador, incluindo discos e fitas.

Memória Virtual (Virtual Memory Storage)* Espaço de armazenamento que pode ser visto como o armazenamento principal endereçável pelo usuário do computador, no qual um endereço virtual é convertido em um endereço real. O tamanho da memória virtual é limitado pelo esquema de endereçamento do computador e pela quantidade de armazenamento auxiliar disponível, e não pelo número de posições da memória principal.

Memória Volátil (Volatile Memory) Memória que requer suprimento de energia constante para manter o conteúdo da memória. Se a energia é desligada, a informação armazenada é perdida.

Mestre de Barramento (Bus Master) Dispositivo conectado ao barramento que é capaz de iniciar e controlar uma comunicação através do barramento.

Microcomputador (Microcomputer)* Sistema de computador cuja unidade de processamento é um microprocessador. Um microcomputador básico inclui um microprocessador, memória e facilidades de E/S, que podem ou não estar na mesma pastilha.

Microinstrução (Microinstruction)* Instrução que controla o fluxo e o seqüenciamento de dados em um processador, em nível mais fundamental que o de instruções de máquina. Instruções de máquina individuais e talvez outras funções podem ser implementadas por microprogramas.

Microoperação (Micro-Operation) Operação elementar da CPU, efetuada durante um pulso de relógio.

Microprocessador (Microprocessor)* Processador cujos elementos foram reduzidos a um ou a alguns circuitos integrados.

Microprograma (Microprogram)* Uma seqüência de microinstruções, armazenadas em uma memória especial, onde podem ser acessadas dinamicamente, para efetuar várias funções.

Multiplexador (Multiplexer) Circuito combinatório que conecta múltiplas entradas a uma única saída. Em qualquer instante, apenas uma das entradas é selecionada e passada para a saída.

Multiprocessador (Multiprocessor)* Computador com dois ou mais processadores, que têm acesso a uma área de memória comum.

Mutiprocessador com Acesso Não-Uniforme à Memória (NUMA — Nonuniform Memory Access Multiprocessor) Multiprocessador com memória compartilhada, no qual o tempo de acesso de um processador a uma palavra da memória varia de acordo com a posição dessa palavra na memória.

Multiprocessador Simétrico (SMP — Symmetric Multiprocessing) Forma de multiprocessamento que permite ao sistema operacional executar em qualquer processador disponível, dentre um conjunto de processadores disponíveis simultaneamente.

Multiprogramação (Multiprogramming)* Modo de operação que provê execução intercalada de dois ou mais programas de computador por um único processador.

Núcleo (Nucleus) Porção do sistema operacional que contém suas funções mais básicas e mais freqüentente usadas. Muitas vezes, o núcleo permanece residente na memória principal.

Opcode Forma abreviada de *operation code*.

Operador Binário (Binary Operator)* Operador que representa uma operação com dois e apenas dois operandos.

Operador Unário (Unary Operator)* Operador que representa uma operação com exatamente um operando.

Operando (Operand)* Entidade sobre a qual é efetuada uma operação.

Ortogonalidade (Orthogonality) Princípio pelo qual duas variáveis ou dimensões são independentes uma da outra. No contexto de conjunto de instruções, o termo geralmente é usado para indicar que outros elementos da instrução (modo de endereçamento, número de operandos, tamanho de operando) são independentes do (não determinados por) código de operação.

Pacote de Discos (Disk Pack)* Conjunto de discos magnéticos que podem ser removidos como um todo de uma unidade de disco, juntamente com sua embalagem, da qual os discos devem ser separados quando em operação.

Página (Page)* Em sistemas de memória virtual, um bloco de dados de tamanho fixo, que tem um endereço virtual, e que é transferido como uma unidade entre a memória real e a memória auxiliar.

Paginação sob Demanda (Demand Paging)* Método de paginação em que uma página é transferida da memória auxiliar para a memória principal apenas no momento em que é requerida.

Palavra de Estado de Programa (PSW — Program Status Word) Área de memória usada para indicar a ordem em que as instruções são executadas e para manter o estado do sistema de computador. Sinônimo de *palavra de estado do processador* (processor status word).

Palavra de Instrução Muito Longa (VLIW — Very Long Instruction Word) Refere-se ao uso de instruções que contêm múltiplas operações. De fato, múltiplas instruções estão contidas em uma única palavra. Tipicamente, uma VLIW é construída pelo compilador, que coloca instruções que podem ser executadas em paralelo na mesma palavra.

Pilha (Stack)* Lista construída e mantida de modo que o próximo item a ser recuperado é o item que foi armazenado mais recentemente na lista; o último a entrar é o primeiro a sair (LIFO — Last In First Out).

Pipeline (*Pipeline*) Organização de processador na qual o processador consiste de vários estágios, possibilitando que múltiplas instruções sejam executadas de forma concorrente.

Porta Lógica (Gate) Circuito eletrônico que produz uma saída que é igual ao resultado de uma operação booleana simples sobre seus sinais de entrada.

Previsão de Desvio (Branch Prediction) Mecanismo usado pelo processador para prever o resultado de uma instrução de desvio antes que ela seja executada.

Processador (Processor)* Em um computador, é a unidade funcional que interpreta e executa instruções. Um processador consiste de, pelo menos, uma unidade de controle e uma unidade lógica e aritmética.

Processador com Superpipeline (Superpipelined Processor) Projeto de processador no qual a *pipeline* de instruções consiste de vários estágios muito pequenos, de modo que mais de um estágio da *pipeline* pode ser executado durante um ciclo de relógio, possibilitando que um grande número de instruções possa estar presente na *pipeline* ao mesmo tempo.

Processador Superescalar (Superscalar Processor) Projeto de processador que inclui *pipelines* de múltiplas instruções, de modo que mais de uma instrução pode estar sendo executada no mesmo estágio da *pipeline* simultaneamente.

Processo (Process) Programa em execução. Um processo é selecionado para execução e controlado pelo sistema operacional.

Protocolo de Coerência de Cache (Cache Coherence Protocol) Mecanismo para manter a validade de dados armazenados em várias caches, de modo que qualquer acesso aos dados armazenados nas caches sempre resulte na obtenção da versão mais recente do conteúdo de uma palavra da memória principal.

RAM Dinâmica (Dynamic RAM) Memória RAM cujas células são implementadas usando capacitores. Uma RAM dinâmica perde gradualmente os dados nela armazenados, a não ser que sejam periodicamente restabelecidos.

RAM Estática (Static RAM) Memória RAM cujas células são implementadas usando flip-flops. Uma RAM estática mantém seus dados enquanto tiver suprimento de energia, não requerendo restabelecimento periódico dos dados.

Registrador de Endereço de Instrução (Instruction Address Register)* Registrador de propósito especial, usado para manter o endereço da próxima instrução a ser executada.

Registrador de Endereçamento à Memória (MAR — Memory Address Register)* Registrador, em uma unidade de processamento, que contém o endereço da posição de memória que está sendo usada.

Registrador de Instrução (Instruction Register)* Registrador utilizado para armazenar uma instrução a ser interpretada.

Registrador de Propósito Geral (General-Purpose Register)* Registrador, usualmente endereçável explicitamente, de um conjunto de registradores que podem ser usados para diferentes propósitos (por exemplo, como acumulador, registrador índice ou para manipulação especial de dados).

Registrador Índice (Index Register)* Registrador cujo conteúdo é usado para modificar um endereço de operando durante a execução de uma instrução de computador; também pode ser usado como contador. Um registrador índice pode ser usado para controlar a execução de um laço de repetição, o acesso a elementos de um vetor, como uma chave, para o endereçamento de tabelas, ou como um apontador.

Registrador Temporário de Dados (MBR — Memory Buffer Register) Registrador que contém um dado lido da memória ou a ser escrito na memória.

Registradores (Registers) Memória interna rápida da CPU. Alguns registradores são visíveis para o usuário; isto é, são disponíveis para o programador via conjunto de instruções de máquina. Outros registradores são usados apenas pela CPU, para funções de controle.

Registradores de Controle (Control Registers) Registradores da CPU usados para controlar a operação da CPU. A maioria desses registradores não é visível para o usuário.

Registradores Visíveis para o Usuário (User-Visible Registers) Registradores da CPU que podem ser referenciados pelo programador. O formato do conjunto de instruções possibilita especificar um ou mais registradores como operandos ou endereços de operandos.

Representação em Complemento de Dois (Twos Complement Representation) Representação de números inteiros binários, em que um número inteiro positivo é representado tal como na representação sinal-magnitude. Um número inteiro negativo é representado adicionando 1 ao padrão de bits obtido pela representação em complemento de um desse número.

Representação em Complemento de Um (Ones Complement Representation) Usada para representar números inteiros binários. Um número inteiro positivo é representado como na representação de sinal-magnitude. Um número inteiro negativo é representado invertendo cada bit da representação do número inteiro positivo de mesma magnitude.

Representação Sinal-magnitude (Sign-Magnitude Representation) Usada para representar números binários inteiros. Em uma palavra de N bits, o bit mais à esquerda é o bit de sinal (0 = positivo, 1 = negativo) e os demais N-1 bits representam a magnitude do número.

Semicondutor (Semiconductor) Substância sólida cristalina, tal como silício ou germânio, cuja condutividade elétrica é intermediária entre a de materiais isolantes e bons condutores. É usado para fabricar transistores e componentes de estado sólido.

Sistema de Representação de Ponto Fixo (Fixed-Point Representation System)* Sistema de representação de números fracionários em que a vírgula decimal que indica a parte fracionária do número tem posição fixa implícita na seqüência de dígitos que representa o número, sendo essa posição fixada por alguma convenção.

Sistema de Representação de Ponto Flutuante (Floating-Point Representation System)* Sistema de representação de números reais, em que os números são representados por um par de numerais distintos. O número real denotado é igual ao produto de um desses numerais (parte de ponto fixo) pelo valor obtido elevando-se a base do sistema de numeração (implícita) ao expoente representado pelo segundo numeral da representação (em ponto flutuante).

Sistema Operacional (Operating System)* Programa que controla a execução de programas e provê serviços tais como alocação de recursos, escalonamento de tarefas, controle de entrada/saída e gerenciamento de dados.

Tabela Verdade (Truth Table)* Tabela que descreve uma função lógica, listando todas as possíveis combinações de valores de entrada e indicando, para cada combinação, o valor de saída correspondente.

Tempo de Ciclo de Memória (Memory Cycle Time) Inverso da taxa de acesso a dados na memória. É o menor tempo decorrido entre a resposta a uma requisição de acesso à memória (leitura ou escrita) e a resposta à próxima requisição de acesso.

Tempo de Ciclo do Processador (Processor Cycle Time) Tempo requerido para a execução da menor microoperação da CPU. É a unidade básica de tempo para medir todas as ações da CPU. Sinônimo de *tempo de ciclo de máquina*.

Temporização Assíncrona (Asynchronous Timing) Técnica na qual a ocorrência de um evento no barramento segue e depende da ocorrência de um evento anterior.

Temporização Síncrona (Synchronous Timing) Técnica na qual a ocorrência de eventos no barramento é determinada por um relógio. O relógio define intervalos de tempo iguais e os eventos iniciam apenas no início de cada um desses intervalos de tempo.

Unidade Central de Processamento (CPU — Central Processing Unit) É a parte do computador que busca e executa instruções. Consiste de uma unidade lógica e aritmética (ULA), uma unidade de controle e registradores. Freqüentemente denominada de forma mais simples como *processador*.

Unidade de Controle (Control Unit) Parte da CPU que controla a execução de operações na CPU, incluindo operações da ULA, movimentação de dados dentro da CPU e troca de dados e sinais de controle por meio de interfaces externas (por exemplo, através do barramento de sistema).

Unidade Lógica e Aritmética (ULA, Arithmetic and Logic Unit — ALU)* É a parte do computador que efetua operações aritméticas, lógicas e outras operações relacionadas.

Uniprocessamento (Uniprocessing) Execução seqüencial de instruções pela unidade de processamento, ou uso independente da unidade de processamento em um sistema de multiprocessamento.

Variável Global (Global Variable) Variável definida em uma porção de um programa de computador e usada pelo menos em outra porção desse programa.

Variável Local (Local Variable) Uma variável que é definida e usada apenas em uma porção de um programa de computador.

Vetor (Vector)* Quantidade usualmente caracterizada por um conjunto ordenado de dados escalares.

WORM (Write-Once, Read-Many — Escrita Única, Várias Leituras) Disco em que a operação de escrita é feita mais facilmente que no CD-ROM, tornando a produção de uma única cópia do disco viável comercialmente. Assim como o CD-ROM, depois de efetuada a operação de escrita, o disco apenas pode ser usado para leitura. O tamanho mais popular é de 5,25 polegadas e pode armazenar de 200 a 800 Mbytes de dados.

Referências Bibliográficas

[Acosta, 1986] Acosta, R.; Kjelstrup, J. e Torng, H. "An instruction issuing approach to enhancing performance in multiple functional unit processors", *IEEE Transactions on Computers*, set. 1986.

[Adamek, 1991] Adamek, J. *Foundations of coding*. New York: Wiley, 1991.

[Agarwal, 1989] Agarwal, A. *Analysis of cache performance for operating systems and multiprogramming*. Boston: Kluwer Academic Publishers, 1989.

[Agerwala, 1987] Agerwala, T. e Cocke, J. *High performance reduced instruction set processors*. Relatório técnico RC12434 (#55845). Yorktown: IBM Thomas J. Watson Research Center, jan. 1987.

[Alexandridis, 1993] Alexandridis, N. *Design of microprocessor-based systems*. Englewood Cliffs: Prentice Hall, 1993.

[Anderson, 1967] Anderson, D.; Sparacio, F. e Tomasulo, F. "The IBM system/360 model 91: machine philosophy and instruction handling", *IBM Journal of Research and Development*, jan. 1967.

[Anderson, 1998a] Anderson, D. e Shanley, T. *Pentium Pro and Pentium II system architecture*. Reading: Addison-Wesley, 1998.

[Anderson, 1998b] Anderson, D. *FireWire system architecture*. Reading: Addison-Wesley, 1998.

[Atkins, 1996] Atkins, M. "PC software performance tuning", *IEEE Computer*, ago. 1996.

[Azimi, 1992] Azimi, M.; Prasad, B. e Bhat, K. "Two level cache architectures", *Proceedings COMPCON '92*, fev. 1992.

[Baentsch, 1997] Baentsch, M. et al. "Enhancing the Web's infrastructure: from caching to replication", *Internet Computing*, mar./abr. 1997.

[Bashe, 1981] Bashe, C.; Bucholtz, W.; Hawkins, G.; Ingram, J. e Rochester, N. "The architecture of IBM's early computers", *IBM Journal of Research and Development*, set. 1981.

[Bashteen, 1991] Bashteen, A.; Lui, I. e Mullan, J. "A superpipeline approach to the MIPS architecture", *Proceedings, COMPCON Spring '91*, fev. 1991.

[Bell, 1970] Bell, C.; Cady, R.; McFarland, H.; Delagi, B.; O'Loughlin, J. e Noonan, R. "A new architecture for minicomputers: the DEC PDP-11", *Proceedings, Spring Joint Computer Conference*, 1970.

[Bell, 1971] Bell, C. e Newell, A. *Computer structures: readings and examples*. New York: McGraw-Hill, 1971.

[Bell, 1978a] Bell, C.; Mudge, J. e McNamara, J. *Computer engineering: a DEC view of hardware systems design*. Bedford: Digital Press, 1978.

[Bell, 1978b] Bell, C.; Newell, A. e Siewiorek, D. "Structural levels of the PDP-8". In: Bell (1978a).

[Bell, 1978c] Bell, C.; Kotok, A.; Hastings, T. e Hill, R. "The evolution of the DEC system-10", *Communications of the ACM,* jan. 1978.

[Benham, 1992] Benham, J. "A geometric approach to presenting computer representations of integers", *SIGCSE Bulletin,* dez. 1992.

[Betker, 1997] Betker, M.; Fernando, J. e Whalen, S. "The history of the microprocessor", *Bell Labs Technical Journal,* outono 1997.

[Blaauw, 1997] Blaauw, G. e Brooks, F. *Computer architecture: concepts and evolution.* Reading: Addison-Wesley, 1997.

[Blahut, 1983] Blahut, R. *Theory and practice of error control codes.* Reading: Addison-Wesley, 1983.

[Bohr, 1998] Bohr, M. "Silicon trends and limits for advanced microprocessors", *Communications of the ACM,* mar. 1998.

[Bondurant, 1994] Bondurant, D. "Low latency EDRAM main memory subsystem for 66 Mhz bus operation", *Proceedings, COMPCON '94,* mar. 1994.

[Bradlee, 1991a] Bradlee, D.; Eggers, S. e Henry, R. "The effect on RISC performance of register set size and structure *versus* code generation strategy", *Proceedings, 18th Annual International Symposium on Computer Architecture,* maio 1991.

[Bradlee, 1991b] Bradlee, D.; Eggers, S. e Henry, R. "Integrating register allocation and instruction scheduling for RISCs", *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* abr. 1991.

[Brewer, 1997] Brewer, E. "Clustering: multiply and conquer", *Data Communications,* jul. 1997.

[Brey, 1997] Brey, B. *The Intel microprocessors: 8086/8066, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Processor.* Upper Saddle River: Prentice Hall, 1997.

[Burger, 1997] Burger, D. e Austin, T. "The SimpleScalar tool set, version 2.0", *Computer Architecture News,* jun. 1997.

[Burks, 1946] Burks, A.; Goldstine, H. e Von Neumann, J. *Preliminary discussion of the logical design of an electronic computer instrument.* Relatório preparado para o U.S. Army Ordnance Dept., 1946, reimpresso em Bell (1971a).

[Carter, 1996] Carter, J. *Microprocessor architecture and microprogramming.* Upper Saddle River: Prentice Hall, 1996.

[Catanzaro, 1994] Catanzaro, B. *Multiprocessor system architectures.* Mountain View: Sunsoft Press, 1994.

[Chaitin, 1982] Chaitin, G. "Register allocation and spilling via graph coloring", *Proceedings, SIGPLAN Symposium on Compiler Construction,* jun. 1982.

[Chen, 1994] Chen, P.; Lee, E.; Gibson, G.; Katz, R. e Patterson, D. "RAID: high-performance, reliable secondary storage", *ACM Computing Surveys,* jun. 1994.

[Chow, 1986] Chow, F.; Himmelstein, M.; Killian, E. e Weber, L. "Engineering a RISC compiler system", *Proceedings, COMPCON Spring '86,* mar. 1986.

[Chow, 1987] Chow, F.; Correll, S.; Himmelstein, M.; Killian, E. e Weber, L. "How many addressing modes are enough?", *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems,* out. 1987.

[Chow, 1990] Chow, F. e Hennessy, J. "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages,* out. 1990.

[Clark, 1985] Clark, D. e Emer, J. "Performance of the VAX-11/780 translation buffer: simulation and measurement", *ACM Transactions on Computer Systems*, fev. 1985.

[Cohen, 1981] Cohen, D. "On holy wars and a plea for peace", *Computer*, out. 1981.

[Colwell, 1985a] Colwell, R.; Hitchcock, C.; Jensen, E.; Brinkley-Sprunt, H. e Kollar, C. "Computers, complexity, and controversy", *Computer*, set. 1985.

[Colwell, 1985b] Colwell, R.; Hitchcock, C.; Jensen, E. e Sprunt, H. "More controversy about 'computers, complexity, and controversy'", *Computer*, dez. 1985.

[Comerford, 1995] Comerford, R. "An overview of high performance", *IEEE Spectrum*, abr. 1995.

[Cook, 1982] Cook, R. e Dande, N. "An experiment to improve operand addressing", *Proceedings, Symposium on Architecture Support for Programming Languages and Operating Systems*, mar. 1982.

[Coonen, 1981] Coonen J. "Underflow and denormalized numbers", *IEEE Computer*, mar. 1981.

[Coutant, 1986] Coutant, D.; Hammond, C. e Kelley, J. "Compilers for the new generation of Hewlett-Packard computers", *Proceedings, COMPCON Spring '86*, mar. 1986.

[Cragon, 1979] Cragon, H. "An evaluation of code space requirements and performance of various architectures", *Computer Architecture News*, fev. 1979.

[Crawford, 1990] Crawford, J. "The i486 CPU: executing instructions in one clock cycle", *IEEE Micro*, fev. 1990.

[Cragon, 1992] Cragon, H. *Branch strategy taxonomy and performance models*. Los Alamitos: IEEE Computer Society Press, 1992.

[Crisp, 1997] Crisp, R. "Direct RAMBUS technology: the new main memory standard", *IEEE Micro*, nov./dez. 1997.

[Dattatreya, 1993] Dattatreya, G. "A systematic approach to teaching binary arithmetic in a first course", *IEEE Transactions on Education*, fev. 1993.

[Davidson, 1987] Davidson, J. e Vaughan, R. "The effect of instruction set complexity on program size and memory performance", *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems*, out. 1987.

[Denning, 1968] Denning, P. "The working set model for program behavior", *Communications of the ACM*, maio 1968.

[Dewar, 1990] Dewar, R. e Smosna, M. *Microprocessors: a programmer's view*. New York: McGraw-Hill, 1990.

[Diefendorff, 1994] Diefendorff, K.; Oehler, R. e Hochsprung, R. "Evolution of the PowerPC architecture", *IEEE Micro*, abr. 1994.

[Dizkstra, 1963] Dizkstra, E. "Making an ALGOL translator for the XI", *Annual Review of Automatic Programming*, v. 4, Pergamon, 1963.

[Doetting, 1997] Doetting, G. et al. "S/390 parallel enterprise server generation 3: a balanced system and cache structure", *IBM Journal of Research and Development*, jul./set. 1997.

[Dubey, 1991] Dubey, P. e Flynn, M. "Branch strategies: modeling and optimization", *IEEE Transactions on Computers*, out. 1991.

[Dulong, 1998] Dulong, C. "The IA-64 architecture at work", *Computer*, jul. 1998.

[Erkert, 1990] Erkert, R. "Communication between computers and peripherd devices: an analogy", *ACM SIGCSE Bulletin*, set. 1990.

[El-Ayat, 1985] El-Ayat, K. e Agarwal, R. "The Intel 80386: architecture and implementation", *IEEE Micro*, dez. 1985.

[Evers, 1998] Evers, M. et al. "An analysis of correlation and predictability: what makes two-level branch predictors work", *Proceedings, 25th Annual International Symposium on Microarchitecture*, jul. 1998.

[Fitzpatrick, 1981] Fitzpatrick, D. et al. "A RISCy approach to VLSI", *VLSI Design*, 4th quarter, 1981. Reimpresso em *Computer Architecture News*, mar. 1982.

[Flynn, 1971] Flynn, M. e Rosin, R. "Microprogramming: an introduction and a viewpoint", *IEEE Transactions on Computers*, jul. 1971.

[Flynn, 1972] Flynn, M. "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, set. 1972.

[Flynn, 1985] Flynn, M.; Johnson, J. e Wakefield, S. "On instruction sets and their formats", *IEEE Transactions on Computers*, mar. 1985.

[Flynn, 1987] Flynn, M.; Mitchell, C. e Mulder, J. "And now a case for more complex instruction sets", *Computer*, set. 1987.

[Frailey, 1983] Frailey, D. "Word length of a computer architecture: definitions and applications", *Computer Architecture News*, jun. 1983.

[Friedman, 1996] Friedman, M. "RAID keeps going and going and ...", *IEEE Spectrum*, abr. 1996.

[Furht, 1987] Furht, B. e Milutinovic, V. "A survey of microprocessor architectures for memory management", *Computer*, mar. 1987.

[Garrett, 1994] Garrett, B. "RDRAMs: a new speed paradigm", *Proceedings, COMPCON '94*, mar. 1994.

[Gehringer, 1988] Gehringer, E.; Abullarade, J. e Gulyn, M. "A survey of commercial parallel processors", *Computer Architecture News*, set. 1988.

[Gifford, 1987] Gifford, D. e Spector, A. "Case study: IBM's system/360-370 architecture", *Communications of the ACM*, abr. 1987.

[Gillingham, 1997] Gillingham, P. e Vogley, B. "SLDRAM: high-performance open-standard memory", *IEEE Micro*, nov./dez. 1997.

[Gjessing, 1992] Gjessing, S. et al. "A RAM link for high speed", *IEEE Spectrum*, out. 1992.

[Goldberg, 1991] Goldberg, D. "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys*, mar. 1991. Disponível em http://www.validgh.com/

[Goor, 1989] Goor, A. *Computer architecture and design*. Reading: Addison-Wesley, 1989.

[Halfhill, 1997] Halfhill, T. "Beyond Pentium II", *Byte*, dez. 1997.

[Handy, 1998] Handy, J. *The cache memory book*. San Diego: Academic Press, 1998.

[Hayes, 1988] Hayes, J. *Computer architecture and organization*. Nova York: McGraw-Hill, 1988.

[Heath, 1984] Heath, J. "Re-evaluation of RISC I", *Computer Architecture News*, mar. 1984.

[Hennessy, 1982] Hennessy, J. et al. "Hardware/software tradeoffs for increased performance", *Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems*, mar. 1982.

[Hennessy, 1984] Hennessy, J. "VLSI processor architecture", *IEEE Transactions on Computers*, dez. 1984.

[Hennessy, 1991] Hennessy, J. e Jouppi, N. "Computer technology and architecture: an evolving interaction", *Computer*, set. 1991.

[Hennessy, 1996] Hennessy, J. e Patterson, D. *Computer architecture: a quantitative approach.* San Mateo: Morgan Kaufmann, 1996.

[Hidaka, 1990] Hidaka, H.; Matsuda, Y.; Asakura, M. e Kazuyasu, F. "The cache DRAM architecture: a DRAM with an on-chip cache memory", *IEEE Micro,* abr. 1990.

[Higbie, 1990] Higbie, L. "Quick and easy cache performance analysis", *Computer Architecture News,* jun. 1990.

[Hill, 1964] Hill, R. "Stored logic programming and applications", *Datamation,* fev. 1964.

[Hill, 1989] Hill, M. "Evaluating associativity in CPU caches", *IEEE Transactions on Computers,* dez. 1989.

[Hewlett-Packard, 1996] Hewlett-Packard. *White paper on clustering.* Disponível em http://www.hp.com/netserver/partners/papers/wp_clust_696.html, jun. 1996.

[Huck, 1983] Huck, T. *Comparative analysis of computer architectures.* Relatório técnico da Universidade de Stanford n. 83-243, maio 1983.

[Huguet, 1991] Huguet, M. e Lang, T. "Architectural support for reduced register saving/restoring in single-window register files", *ACM Transactions on Computer Systems,* fev. 1991.

[Hutcheson, 1996] Hutcheson, G. e Hutcheson, J. "Technology and economics in the semiconductor industry", *Scientific American,* jan. 1996.

[Hwang, 1993] Hwang, K. *Advanced computer architecture.* New York: McGraw-Hill, 1993.

[Hwu, 1998] Hwu, W. "Introduction to predicated execution", *Computer,* jan. 1998.

[IBM, 1994] International Business Machines, Inc. *The PowerPC architecture: a specification for a new family of RISC processors.* San Francisco: Morgan Kaufmann, 1994.

[IEEE, 1985] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985, 1985.

[Intel, 1998a] Intel Corp. *Pentium processors and related products.* Aurora, 1998.

[Intel, 1998b] Intel Corp. *Pentium Pro and Pentium II processors and related products.* Aurora, 1998.

[James, 1990] James, D. "Multiplexed buses: the endian wars continue", *IEEE Micro,* set. 1990.

[Johnson, 1991] Johnson, M. *Superscalar microprocessor design.* Englewood Cliffs: Prentice Hall, 1991.

[Jouppi, 1988] Jouppi, N. "Superscalar *versus* superpipelined machines", *Computer Architecture News,* jun. 1988.

[Jouppi, 1989a] Jouppi, N. e Wall, D. "Available instruction-level parallelism for superscalar and superpipelined machines", *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems,* abr. 1989.

[Jouppi, 1989b] Jouppi, N. "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance", *IEEE Transactions on Computers,* dez. 1989.

[Kaeli, 1991] Kaeli, D. e Emma, P. "Branch history table prediction of moving target branches due to subroutine returns", *Proceedings, 18th Annual International Symposium on Computer Architecture,* maio 1991.

[Kane, 1992] Kane, G. e Heinrich, J. *MIPS RISC architecture.* Englewood Cliffs: Prentice Hall, 1992.

[Katevenis, 1983] Katevenis, M. *Reduced instruction set computer architectures for VLSI.* Tese de doutorado, Departamento de Ciência da Computação, Universidade da Califórnia, Berkeley, out. 1983. Reimpresso pela MIT Press, Cambridge, 1985.

[Katz, 1989] Katz, R.; Gibson, G. e Patterson, D. "Disk system architecture for high performance computing", *Proceedings of the IEEE,* dez. 1989.

[Knuth, 1971] Knuth, D. "An empirical study of FORTRAN programs", *Software Practice and Experience,* v. 1, 1971.

[Knuth, 1998] Knuth, D. *The art of computer programming, volume 2: Seminumerical algorithms.* Reading: Addison-Wesley, 1998.

[Koren, 1993] Koren, I. *Computer arithmetic algorithms.* Englewood Cliffs: Prentice Hall, 1993.

[Kuck, 1972] Kuck, D.; Muraoka, Y. e Chen, S. "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup", *IEEE Transactions on Computers,* dez. 1972.

[Kuga, 1991] Kuga, M.; Murakami, K. e Tomita, S. "DSNS (Dynamically-hazard resolved, Statically-code-scheduled, Nonuniform Superscalar): yet another superscalar processor architecture", *Computer Architecture News,* jun. 1991.

[Lee, 1991] Lee, R.; Kwok, A. e Briggs, F. "The floating point performance of a superscalar SPARC processor", *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* abr. 1991.

[Lilja, 1988] Lilja, D. "Reducing the branch penalty in pipelined processors", *Computer,* jul. 1988.

[Lilja, 1993] Lilja, D. "Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons", *ACM Computing Surveys,* set. 1993.

[Lovett, 1996] Lovett, T. e Clapp, R. "Implementation and performance of a CC-NUMA system", *Proceedings, 23rd Annual International Symposium on Computer Architecture,* maio 1996.

[Lunde, 1977] Lunde, A. "Empirical evaluation of some features of instruction set processor architectures", *Communications of the ACM,* mar. 1977.

[Lynch, 1993] Lynch, M. *Microprogrammed state machine design.* Boca Raton: CRC Press, 1993.

[MacGregor, 1984] MacGregor, D.; Mothersole, D. e Moyer, B. "The Motorola MC68020", *IEEE Micro,* ago. 1984.

[Mahlke, 1994] Mahlke, S. et al. "Characterizing the impact of predicated execution on branch prediction" *Proceedings, 27th International Symposium on Microarchitecture,* dez. 1994.

[Mahlke, 1995] Mahlke, S. et al. "A comparison of full and partial predicated execution support for ILP processors", *Proceedings, 22nd International Symposium on Computer Architecture,* jun. 1995.

[Mak, 1997] Mak, P. et al. "Shared-cache clusters in a system with a fully shared memory", *IBM Journal of Research and Development,* jul./set. 1997.

[Mallach, 1979] Mallach, E. "The evolution of an architecture", *Datamation,* abr. 1979.

[Mallach, 1975] Mallach, E. "Emulation architecture", *Computer,* ago. 1975.

[Mallach, 1983] Mallach, E. e Sondak, N. *Advances in microprogramming.* Dedham: Artech House, 1983.

[Mano, 1997] Mano, M. e Kime, C. *Logic and computer design fundamentals.* Upper Saddle River: Prentice Hall, 1997.

[Mansuripur, 1997] Mansuripur, M. e Sincerbox, G. "Principles and techniques of optical data storage", *Proceedings of the IEEE,* nov. 1997.

[Marchant, 1990] Marchant, A. *Optical recording.* Reading: Addison-Wesley, 1990.

[Mashey, 1995] Mashey, J. "CISC *vs.* RISC (or what is RISC really)", *USENET comp.arch newsgroup, article 46782,* fev. 1995.

[Massiglia, 1997] Massiglia, P. *The RAID book: a storage system technology handbook.* St. Peter: The Raid Advisory Board, 1997.

[Mayberry, 1984] Mayberry, W. e Efland, G. "Cache boosts multiprocessor performance", *Computer Design,* nov. 1984.

[McEliece, 1985] McEliece, R. "The reliability of computer memories", *Scientific American,* jan. 1985.

[Mee, 1996a] Mee, C. e Daniel, E. (eds.). *Magnetic recording technology.* New York: McGraw-Hill, 1996.

[Mee, 1996b] Mee, C. e Daniel, E. (eds.). *Magnetic storage handbook.* New York: McGraw-Hill, 1996.

[Mirapuri, 1992] Mirapuri, S.; Woodacre, M. e Vasseghi, N. "The MIPS R4000 processor", *IEEE Micro,* abr. 1992.

[Moore, 1965] Moore, G. "Cramming more components onto integrated circuits", *Electronics Magazine,* 19 abr. 1965.

[Morgan, 1992] Morgan, D. *Numerical methods.* San Mateo: M&T Books, 1992.

[Morse, 1978] Morse, S.; Pohlman, W. e Ravenel, B. "The Intel 8086 microprocessor: a 16-bit evolution of the 8080", *Computer,* jun. 1978.

[Motorola, 1997] Motorola, Inc. *PowerPC microprocessor family: the programming environments for 32-bit microprocessors.* Denver, 1997.

[Myers, 1978] Myers, G. "The evaluation of expressions in a storage-to-storage architecture", *Computer Architecture News,* jun. 1978.

[Nayfeh, 1996] Nayfeh, B.; Olukotun, K. e Singh, J. "The impact of shared cache clustering in small-scale shared-memory multiprocessors", *Proceedings of the Second International Symposium on High Performance Computer Architecture,* 1996.

[NCR, 1990] NCR Corp. *SCSI: understanding the small computer system interface.* Englewood Cliffs: Prentice Hall, 1990.

[Normoyle, 1998] Normoyle, K. et al. "UltraSPRAC-IIi: expanding the boundaries of a system on a chip", *IEEE Micro,* mar./abr. 1998.

[Novitsky, 1993] Novitsky, J.; Azimi, M. e Ghaznavi, R. "Optimizing systems performance based on Pentium processors", *Proceedings COMPCON '92,* fev. 1993.

[Oberman, 1997a] Oberman, S. e Flynn, M. "Design issues in division and other floating-point operations", *IEEE Transactions on Computers,* fev. 1997.

[Oberman, 1997b] Oberman, S. e Flynn, M. "Division algorithms and implementations", *IEEE Transactions on Computers,* ago. 1997.

[Omondi, 1994] Omondi, A. *Computer arithmetic systems: algorithms, architecture and implementations.* Englewood Cliffs: Prentice Hall, 1994.

[Padegs, 1981] Padegs, A. "System/360 and beyond", *IBM Journal of Research and Development,* set. 1981.

[Padegs, 1988] Padegs, A.; Moore, B.; Smith, R. e Buchholz, W. "The IBM system/370 vector architecture: design considerations", *IEEE Transactions on Communications,* maio 1988.

[Papworth, 1996] Papworth, D. "Tuning the Pentium Pro microarchitecture", *IEEE Micro,* abr. 1996.

[Parker, 1989] Parker, A. e Hamblen, J. *An introduction to microprogramming with exercises designed for the Texas Instruments SN74ACT8800 software development board.* Dallas: Texas Instruments, 1989.

[Patterson, 1982a] Patterson, D. e Sequin, C. "A VLSI RISC", *Computer*, set. 1982.

[Patterson, 1982b] Patterson, D. e Piepho, R. "Assessing RISCs in high-level language support", *IEEE Micro*, nov. 1982.

[Patterson, 1984] Patterson, D. "RISC watch", *Computer Architecture News*, mar. 1984.

[Patterson, 1985a] Patterson, D. "Reduced instruction set computers", *Communications of the ACM*, jan. 1985.

[Patterson, 1985b] Patterson, D. e Hennessy, J. "Response to 'Computers, complexity, and controversy'", *Computer*, nov. 1985.

[Patterson, 1988] Patterson, D.; Gibson, G. e Katz, R. "A case for redundant arrays of inexpensive disks (RAID)", *Proceedings, ACM SIGMOD Conference of Management of Data*, jun. 1988.

[Patterson, 1998] Patterson, D. e Hennessy, J. *Computer organization and design: the hardware/software interface.* San Mateo: Morgan Kaufmann, 1998.

[Peleg, 1997] Peleg, A.; Wilkie, S. e Weiser, U. "Intel MMX for multimedia PCs", *Communications of the ACM*, jan. 1997.

[Pfister, 1998] Pfister, G. *In search of clusters.* Upper Saddle River: Prentice Hall, 1998.

[Popescu, 1991] Popescu, V. et al. "The metaflow architecture", *IEEE Micro*, jun. 1991.

[Potter, 1994] Potter, T. et al. "Resolution of data and control-flow dependencies in the PowerPC 601", *IEEE Micro*, out. 1994.

[Prince, 1991] Prince, B. *Semiconductor memories.* New York: Wiley, 1991.

[Prince, 1996] Prince, B. *High performance memories: new architecture DRAMs and SRAMs, evolution and function.* New York: Wiley, 1996.

[Przybylski, 1988] Przybylski, S.; Horowitz, M. e Hennessy, J. "Performance trade-offs in cache design", *Proceedings, Fifteenth Annual International Symposium on Computer Architecture*, jun. 1988.

[Przybylski, 1990] Przybylski, S. "The performance impact of block size and fetch strategies", *Proceedings, 17th Annual International Symposium on Computer Architecture*, maio 1990.

[Przybylski, 1994] Przybylski, S. *New DRAM technologies.* Sebastapol: MicroDesign Resources, 1994.

[Radin, 1983] Radin, G. "The 801 minicomputer", *IBM Journal of Research and Development*, maio 1983.

[Ragan-Kelley, 1983] Ragan-Kelley, R. e Clark, R. "Applying RISC theory to a large computer", *Computer Design*, nov. 1983.

[Rauscher, 1980] Rauscher, T. e Adams, P. "Microprogramming: a tutorial and survey of recent developments", *IEEE Transactions on Computers*, jan. 1980.

[Reches, 1998] Reches, S. e Weiss, S. "Implementation and analysis of path history in dynamic branch prediction schemes", *IEEE Transactions on Computers*, ago. 1998.

[Rosch, 1997] Rosch, W. *Winn L. Rosch hardware bible.* Indianápolis: Sams, 1997.

[Satyanarayanan, 1981] Satyanarayanan, M. e Bhandarkar, D. "Design trade-offs in VAX-11 translation buffer organization", *Computer*, dez. 1981.

[Schaller, 1997] Schaller, R. "Moore's law: past, present, and future", *IEEE Spectrum*, jun. 1997.

[Schmidt, 1997] Schmidt, F. *The SCSI bus and IDE interface.* Reading: Addison-Wesley, 1997.

[Sebern, 1976] Sebern, M. "A minicomputer-compatible microcomputer system: the DEC LSI-11", *Proceedings of the IEEE,* jun. 1976.

[Segee, 1991] Segee, B. e Field, J. *Microprogramming and computer architecture.* New York: Wiley, 1991.

[Serlin, 1986] Serlin, O. "MIPS, dhrystones, and other tales", *Datamation,* 1º jun. 1986.

[Shannon, 1938] Shannon, C. "Symbolic analysis of relay and switching circuits", *AIEE Transactions,* v. 57, 1938.

[Shanley, 1995a] Shanley, T. *PowerPC system architecture.* Reading: Addison-Wesley, 1995.

[Shanley, 1995b] Shanley, T. e Anderson, D. *PCI systems architecture.* Richardson: Mindshare Press, 1995.

[Shanley, 1998] Shanley, T. *Pentium Pro and Pentium II system architecture.* Reading: Addison-Wesley, 1998.

[Sharma, 1997] Sharma, A. *Semiconductor memories: technology, testing, and reliability.* New York: IEEE Press, 1997.

[Sherburne, 1984] Sherburne, R. *Processor design tradeoffs in VLSI.* Tese de doutorado. Relatório n. UCB/CSD 84/173, Universidade da Califórnia, Berkeley, abr. 1984.

[Siewiorek, 1982] Siewiorek, D.; Bell, C. e Newell, A. *Computer structures: principles and examples.* New York: McGraw-Hill, 1982.

[Sima, 1997] Sima, D. "Superscalar instruction issue", *IEEE Micro,* set./out. 1997.

[Simon, 1969] Simon, H. *The sciences of the artificial.* Cambridge: MIT Press, 1969.

[Smith, 1982] Smith, A. "Cache memories", *ACM Computing Surveys,* set. 1982.

[Smith, 1987] Smith, A. "Line (block) size choice for CPU cache memories", *IEEE Transactions on Communications,* set. 1987.

[Smith, 1989] Smith, M.; Johnson, M. e Horowitz, M. "Limits on multiple instruction issue", *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems,* abr. 1989.

[Smith, 1995] Smith, J. e Sohi, G. "The microarchitecture of superscalar processors", *Proceedings of the IEEE,* dez. 1995.

[Soderquist, 1996] Soderquist, P. e Leeser, M. "Area and performance tradeoffs in floating-point divide and square-root implementations", *ACM Computing Surveys,* set. 1996.

[Sohi, 1990] Sohi, G. "Instruction issue logic for high-performance interruptable, multiple functional unit, pipelined computers", *IEEE Transactions on Computers,* mar. 1990.

[Solari, 1994] Solari, E. e Willse, G. *PCI hardware and software: architecture and design.* San Diego: Annabooks, 1994.

[Stallings, 1997] Stallings, W. *Data and computer communications.* 5. ed. Upper Saddle River: Prentice Hall, 1997.

[Stallings, 1998] Stallings, W. *Operating systems, internals and design principles.* 3. ed. Upper Saddle River: Prentice Hall, 1998.

[Stenstrom, 1990] Stenstrom, P. "A survey of cache coherence schemes of multiprocessors", *Computer,* jun. 1990.

[Stevens, 1964] Stevens, W. "The structure of system/360, part II: System implementation", *IBM Systems Journal,* v. 3, n. 2, 1964. Reimpresso em Siewiorek (1982).

[Stone, 1993] Stone, H. *High-performance computer architecture.* Reading: Addison-Wesley, 1993.

[Strecker, 1978] Strecker, W. "VAX-11/780: a virtual address extension to the DEC PDP-11 family", *Proceedings, National Computer Conference*, 1978.

[Strecker, 1983] Strecker, W. "Transient behavior of cache memories", *ACM Transactions on Computer Systems*, nov. 1983.

[Stritter, 1979] Stritter, E. e Gunter, T. "A microprocessor architecture for a changing world: the Motorola 68000", *Computer*, fev. 1979.

[Swartzlander, 1990] Swartzlander, E. (ed.). *Computer arithmetic, volumes I and II*. Los Alamitos: IEEE Computer Society Press, 1990.

[Tabak, 1991] Tabak, D. *Advanced microprocessors*. Nova York: McGraw-Hill, 1991.

[Tamir, 1983] Tamir, Y. e Sequin, C. "Strategies for managing the register file in RISC", *IEEE Transactions on Computers*, nov. 1983.

[Tanenbaum, 1978] Tanenbaum, A. "Implications of structured programming for machine architecture", *Communications of the ACM*, mar. 1978.

[Tanenbaum, 1990] Tanenbaum, A. *Structured computer organization*. Englewood Cliffs: Prentice Hall, 1990.

[Thompson, 1994] Thompson, T. e Ryan, B. "PowerPC 620 soars", *Byte*, nov. 1994.

[Texas Instruments, 1990] Texas Instruments Inc. *SN74ACT880 family data manual*. SCSS006C, 1990.

[Tjaden, 1970] Tjaden, G. e Flynn, M. "Detection and parallel execution of independent instructions", *IEEE Transactions on Computers*, out. 1970.

[Tomasevic, 1993] Tomasevic, M. e Milutinovic, V. *The cache coherence problem in shared-memory multiprocessors: hardware solutions*. Los Alamitos: IEEE Computer Society Press, 1993.

[Toong, 1981] Toong, H. e Gupta, A. "An architectural comparison of contemporary 16-bit microprocessors", *IEEE Micro*, maio 1981.

[Tremblay, 1996] Tremblay, M. et al. "VIS speeds new media processing", *IEEE Micro*, ago. 1996.

[Tucker, 1967] Tucker, S. "Microprogram control for system/360", *IBM Systems Journal*, n. 4, 1967.

[Tucker, 1987] Tucker, S. "The IBM 3090 system design with emphasis on the vector facility", *Proceedings, COMPCON Spring '87*, fev. 1987.

[Voelker, 1988] Voelker, J. "The PDP-8", *IEEE Spectrum*, nov. 1988.

[Vogley, 1994] Vogley, B. "800 megabyte per second systems via use of synchronous DRAM", *Proceedings, COMPCON '94*, mar. 1994.

[Von Neumann, 1945] Von Neumann, J. *First draft of a report on the EDVAC*. Moore School, University of Pennsylvania, 1945.

[Vranesic, 1980] Vranesic, Z. e Thurber, K. "Teaching computer structures", *Computer*, jun. 1980.

[Wallich, 1985] Wallich, P. "Toward simpler, faster computers", *IEEE Spectrum*, ago. 1985.

[Wallis, 1990] Wallis, P. *Improving floating-point programming*. New York: Wiley, 1992.

[Wall, 1991] Wall, D. "Limits of instruction-level parallelism", *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, abr. 1991.

[Ward, 1990] Ward, S. e Halstead, R. *Computation structures*. Cambridge: MIT Press, 1990.

[Weinberg, 1975] Weinberg, G. *An introduction to general systems thinking*. New York: Wiley, 1975.

[Weiss, 1984] Weiss, S. e Smith, J. "Instruction issue logic in pipelined supercomputers", *IEEE Transactions on Computers*, nov. 1984.

[Weiss, 1994] Weiss, S. e Smith, J. *POWER and PowerPC*. San Francisco: Morgan Kaufmann, 1994.

[Whitney, 1997] Whitney, S. et al. "The SGI origin software environment and application performance", *Proceedings, COMPCON Spring '97*, fev. 1997.

[Wilkes, 1951] Wilkes, M. "The best way to design an automatic calculating machine", *Proceedings, Manchester University Computer Inaugural Conference*, jul. 1951.

[Wilkes, 1953] Wilkes, M. e Stringer, J. "Microprogramming and the design of the control circuits in an electronic digital computer", *Proceedings of the Cambridge Philosophical Society*, abr. 1953. Reimpresso em Siewiorek (1982).

[Williams, 1990] Williams, F. e Steven, G. "Address and data register separation on the M68000 family", *Computer Architecture News*, jun. 1990.

[Wilson, 1984] Wilson, K. "Science, industry, and the new Japanese challenge", *Proceedings of the IEEE*, jan. 1984.

[Yeager, 1996] Yeager, K. "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, abr. 1996.

[Yeh, 1991] Yeh, T. e Patt, Y. "Two-level adaptive training branch prediction", *Proceedings, 24th Annual International Symposium on Microarchitecture*, 1991.

ÍNDICE

A

D

G

H

I

Processadores vetoriais, 686

S

T

U

V

W